REPORT DOCUMENTATION PAGE

AFRL-SR-AR-TR-03-

0076

1. REPORT DATE *(DD-MM-YYYY)*
14-02-2003

2. REPORT TYPE
Final Report

3. DATES COVERED *(From - To)*
01-06-2001 - 31-08-2002

4. TITLE AND SUBTITLE

Faculty Fellowship in Support of Tolerating Intrusions Through Secure System Reconfiguration

5a. CONTRACT NUMBER
F49620-01-1-0282

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

6. AUTHOR(S)

Heimbigner, Dennis M.
Wolf, Alexander L.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Attn. Ralph Brown
Office of Contracts and Grants
University of Colorado
3100 Marine St.Rm.481
Boulder,CO 80309-0572

8. PERFORMING ORGANIZATION REPORT NUMBER
0202.00.0343B

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Attn. Dr. Spencer Wu NJ
Air Force Office of Scientific Research
4015 Wilson Blvd, Room 713
Arlington, VA 22203-1954

10. SPONSOR/MONITOR'S ACRONYM(S)
AFOSR

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Unrestricted

DISTRIBUTION STATEMENT A
Approved for Public Release
Distribution Unlimited

20030326 021

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This fellowship was intended to increase the capabilities of the identified fellow with respect to the area of security and information assurance by placing the fellow at the University of California at Davis Computer Science Department. This process succeeded by a number of measures. The fellow was able to quickly get up-to-speed on security. He identified several possible new lines of research in the areas of untrusted computing, use of secure hardware, and responses to flash worms. He also developed a new use for deception, which has received funding from AFRL. These ideas were presented and critiqued through several colloquium presentations to the security group at UC, Davis. The fellow is also developing a curriculum for a course in undergraduate security using the new textbook by Professor Matt Bishop of UC, Davis. A number of publications also resulted from this fellowship.

15. SUBJECT TERMS
Intrusion Tolerance, Security, Reconfiguration, secure co-processor, anti-tamper, flash worm

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON Dennis Heimbigner
a. REPORT	b. ABSTRACT	c. THIS PAGE			19b. TELEPHONE NUMBER *(include area code)*
U	U	U	UU	36	303-492-6643

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

BEST AVAILABLE COPY

Faculty Fellowship in Support of Tolerating Intrusions Through Secure System Reconfiguration

AFOSR Contract F49620-01-1-0282
Final Report for Period 5/1/2001 – 8/31/2002

02/14/2003

Dennis Heimbigner and Alexander Wolf
Computer Science Department
University of Colorado, Boulder

1 Introduction

The purpose of this fellowship was to increase the capabilities of the identified fellow with respect to three activities: education, curriculum development, and research in the area of security and information assurance. The plan was to place the fellow at the University of California at Davis Computer Science Department. Residence at UC Davis for an extended period of time would allow the fellow to obtain sufficient training to implement a core course in security at the University of Colorado. Equally important, the fellow was to initiate security related research growing out of his background in software and systems and extending the research growing out of the parent proposal. Finally, it was hoped that the fellow would serve as the focal point for introducing security concerns into other courses and into other research programs in the University of Colorado Department of Computer Science.

2 Objectives

The following three objectives are abbreviated versions of the ones described in the proposal.

1. Education: The fellow is expected to increase his knowledge in selected areas of security.
2. Curriculum Development: The fellow will obtain sufficient experience and knowledge to establish and teach at least one core security class at the University of Colorado.
3. Research: The fellow will extend his current research in the direction of security through interaction with researchers at UC Davis, by attending relevant professional meetings, and by developing new research projects in the area of security.

3 Personnel Supported

A specific research faculty person, Dr. Dennis Heimbigner, filled the role of the fellow funded under this proposal.

4 Accomplishments

4.1 Research Accomplishments

The fellow was rapidly able to absorb significant material about security and was able to apply that material to develop some promising new research ideas, one of which as already resulted in external funding. These ideas are briefly described in the following sections.

4.1.1 A Tamper-Detecting Implementation of Lisp

The fellow has developed a novel approach to the problem of carrying out trusted computations in an untrusted environment. This approach combines a secure co-processor with an untrusted host computer. A tamper-detecting language interpreter (for Lisp) executes on the secure co-processor while the code and data of the program reside in the memory of the untrusted host. The term "tamper-detecting" means that any attempt to corrupt a computation carried out by a program in the language will be detected on-line and the computation aborted.

This approach has several advantages including ease of use and the ability to provide tamper-detection for any program that can be constructed using the language. This new approach thus provides a convenient and general mechanism for safe utilization of the memory of an untrusted host by the secure co-processor.

4.1.2 Architectural Degradation against Software Misuse

The fellow has developed an interesting new use for deceptive tactics against the insider problem, where a person with legitimate access to a system utilizes that access to subvert the system. Existing approaches, much like intrusion detection in general, have a significant problem with false alarms. The term "false alarm" means that some person or mechanism has detected a pattern of behavior that appears superficially to be suspect, but upon further investigation, turns out to be, in fact, benign behavior.

False alarms present serious difficulties for security administrators. When the response to the alarm is performed manually, the responder (typically the security/system administrator) can be overwhelmed by large numbers of false alarms and can miss genuine alarms. For automated responses, false alarms can cause overall system operation to seriously degrade as the system thrashes between normal states and intrusion resistant states.

The proposed new approach provides a controlled sequence of responses to potential misuse that allows for better response to false alarms while retaining the ability to disarm (and later rearm) the software by degrees. Thus, false alarms would be met with more measured responses that provided both time and information necessary to verify the correctness of the alarm. As the seriousness of the misuse is ascertained, more severe responses can be brought to bear. In the event of a false alarm, the response sequence and its effects should be reversible so that normal operation is resumed.

The key element of the new approach is to use configuration information to reconfigure the software system into increasingly informative and increasingly degraded modes of operation. By informative, we mean that the component begins to capture and report more detailed information about suspicious activities. Degraded means that the component begins to modify its outputs and actions to reduce its functionality, or in more extreme case, even practice deception and provide wholly or partially false outputs. This latter use of deception represents a new use for the concepts first introduced by Cohen.

The approach also provides two additional capabilities. The first is self-destruction – the ability to reconfigure the software so that it appears to be working but has in fact had all important information and capabilities destroyed, ideally without immediate knowledge by the misuser. The second is restoration – the ability to reconfigure the software to undo degradation responses; this is a unique capability made possible by our use of dynamic reconfiguration, and is important in the event of false alarms or the external neutralizing of the misuse threat.

This work has received new funding through the AFRL Anti-Tamper program (Section 5).

4.1.3 Fast Response Mechanisms Against Fast-Moving Cyber-Attacks

The fellow is leading the development of a project to design, prototype, and evaluate a mechanism for defending large-scale networks against fast-moving malicious attacks such as so-called "flash worms".

Such attacks have the potential to corrupt a large fraction of the Internet within a small period of time. The recent Slammer worm, for example, attacked some 75,00 Internet sites in less than 10 minutes. Such fast moving attacks will use increasingly complex combinations of highly distributed attack modules, operating in parallel and coordinating as necessary to disable targeted sites in the Internet.

The key to defending against such attacks is to "fight fire with fire" by developing an equally fast response mechanism based on four major principles.

1. Subsumption: Local action modified by global concerns. Brook's subsumption concept is adopted, which in this context causes local sensor data to be routed to local agents who rapidly initiate responses to mitigate attacks. These local responses are aggregated and passed to more global agents. These global agents may modify or even suppress local actions in order to achieve better global responses.
2. Dynamic reconfiguration as the primary response mechanism. Responding to an attack requires changing the behavior of the attacked systems. This in turn requires the ability to dynamically reconfigure those systems to achieve the new behavior.
3. Response caching. Fast responses require that the choice of response and the resources needed by that response must be readily available. Thus, various pre-computed resources must be cached at various sites in the network so that they can be readily accessed as needed.
4. Intentional (property-based) communication. The systems to be protected may be numerous and heterogeneous. It is logistically difficult for the defense system to maintain a detailed and complete database of the state of these systems, and so a decentralized intention-based mechanism is needed to communicate with systems based on their current characteristics.

This work extends the parent project: the DARPA-funded Willow project in Intrusion Tolerance. This project is being pursued jointly with Karl Levitt at UC, Davis and John Knight at the University of Virginia.

4.1.4 Hardware-Assisted Intrusion Detection and Response

The fellow is exploring an approach to host-based intrusion detection and response that utilizes secure co-processor hardware. This appears to be an approach that has received little attention in the security community. This approach partially off-loads intrusion detection and response software onto a secure co-processor in order to provide a reliable and unassailable enclave from which the intrusion management software can monitor and control the host in order to detect and repair attacks on the host's software.

Existing (host-based) intrusion detection and response systems (IDRS) generally execute as regular software on the host machine being protected. The intrusion management software is itself a target for attack, which if successful can simultaneously disarm a major security component and provide a fast path for gaining control of the host machine.

Attacks against the co-processor are difficult because it has a restricted functionality and is not providing general services. Its sole purpose is to act as guardian for the associated host. If anomalies are detected, then the co-processor can initiate responses to detect the cause of those anomalies and to repair the system and bring it back to normal operating state.

The primary requirement on the co-processor is that it has unmediated access and control over the host processor. The term “unmediated” is intended to indicate that the host has no ability to stop monitoring, reporting, and repair actions by the co-processor. Examples of unmediated activities include direct read/write access to the main memory of the host, a separate interface to the network, and the ability to forcibly cause an interrupt of the host processor in order to cause it to begin executing some specific code in memory.

If this research is successful, a number of interesting and important capabilities become feasible or more secure; these include detection and repair of attacks, and the ability to provide better capture of forensic evidence for attack analysis and prosecution. The research may also enable a new market for secure co-processors.

4.1.5 Architectural Models in Support of Computer Forensics

The fellow has identified a promising new approach to supporting computer forensic analysis. Such analysis is an important element in overall response to cyber-attacks. When a system crashes, it is critically important to quickly identify the cause of the crash in order to determine if it was caused by an attack. It is then important to identify anomalies indicating the attack mechanism and effects in order to develop a defense against the attack.

Existing analysis tools such as the Coroner’s Toolkit are excellent at providing large quantities of information about the state of a crashed computer, but only at a low level of abstraction provided by the operating system: the processes running and the files open, for example.

What are desperately needed are higher level abstractions that can provide more insight into the state of the system so that a forensic analyst can focus on the anomalies and quickly identify the nature of any attack.

The proposed new approach utilizes various models of software systems to organize and present high level abstractions about the state of crashed systems. The models provide both the “should-be” configuration and the “as-is” configuration. The former represents the range of legal configurations for which a system was designed. The latter represents the apparent configuration of the system as it was at the time of the crash. These representations provide important structural information about the programs that should be, and are, running and how they relate to each other. They provide a high level framework to which other, related information can be attached to provide a growing complex of information about the state of the crashed system. They will also become the basis for presenting the information in ways that will highlight discrepancies between the expected state and the crash state in order to guide the forensic analyst to potential problems.

Over time, this approach can be extended to include data from additional sources and subsequent analyses of that data. We believe that our use of configuration information as the core will produce a forensic analysis system significantly superior to the relatively un-integrated products currently available.

4.2 Education Accomplishments

The fellow began his stay at the University of California at Davis by auditing a number of courses related to security. This continued for the duration of his visit. As a result, the fellow was able to rapidly get up-to-speed on the core security research issues. This was also aided by attending colloquium talks that covered security related problems.

The specific courses audited by the fellow are identified in the following table.

Course No.	Course Title	Instructor	Semester
ECS 227	Modern Cryptography	Rogoway	Fall 2001
ECS 153	Introduction to Computer Security	Bishop	Winter 2002
ECS 289A	Cryptography for E-Commerce	Franklin	Winter 2002
ECS 253	Cryptography and Data Security	Wu	Spring 2002
ECS 289M	Vulnerabilities and Policy Models	Bishop	Spring 2002

4.3 Curriculum Development Accomplishments

An important goal for this fellowship was to allow the fellow to return to Colorado and begin development of at least one core security course. While it is probably unreasonable to suppose that the fellow can teach graduate level courses in security (yet), experience at UC, Davis makes it clear that he should be able to teach a general security course at the undergraduate level.

In line with this goal, the fellow has begun development of a syllabus and materials for that general undergraduate security course. The content of this course is closely patterned after his observations of Professor Matt Bishop at UC, Davis in teaching such a course at Davis. Of equal importance is the fact that the course notes for Professor Bishop's class have recently been released as a textbook entitled "Computer Security: Art and Science" (Addison-Wesley 2002). The fellow intends to use this book as the basis for the course he is developing.

5 Grants Received

1. AFRL, "Architectural Degradation Against Software Misuse", $405,461. 2002-2004.

6 Presentations

The fellow gave four separate colloquium talks to the security group at University of California Davis. The first talk was intended to inform that community about purpose of the fellow's visit and also to inform them about some of the research at the University of Colorado. Subsequent talks were intended to present some potentially new research ideas and to obtain useful feedback.

1. Title: Bend, Don't Break: Using Reconfiguration to Achieve Survivability (December 5, 2001).
2. Title: Intrusion Management Using Architecture and Configuration Models (February 13, 2002)
3. Title: A Tamper-Resistant Programming Language (April 24, 2002)
4. Title: A Pie of P-Baked Security Ideas (June 19, 2002)

Also during the visit to Davis, the fellow participated in three external presentations relevant to the parent project.

1. Demonstrated Willow intrusion tolerance prototype to the Joint Battlespace Infosphere (JBI) group at the Air Force's Rome Labs (7 September 2001).
2. Demonstrated our research at DARPA's DASADA 2002 Exposition and technology demonstration trade show (1-2 July 2002).
3. Invited demonstration at DARPATECH 2002, which is DARPA's premier technology conference (30 July – 1 August 2002)

7 Publications

1. "The Willow Survivability Architecture," John Knight, Dennis Heimbigner, Alexander Wolf, Antonio Carzaniga, Jonathan Hill, and Premkumar Devanbu. Proc. of the 2001 International Survivability Workshop.
2. "A Testbed for Configuration Management Policy Programming," Andre van der Hoek, Antonio Carzaniga, Dennis Heimbigner, and Alexander L. Wolf. IEEE Transactions on Software Engineering 28(1):79–99 (Jan.2002).
3. "A Tamper-Resistant Programming Language," D. Heimbigner, Department of Computer Science Technical Report CU-CS-931-02, University of Colorado, May 2002. Also submitted to SAM'03 as "A Tamper-Detecting Implementation of Lisp."
4. "Intrusion Management Using Configurable Architecture Models," D. Heimbigner and A. Wolf, Department of Computer Science Technical Report CU-CS-929-02, University of Colorado, May 2002.
5. "Reconfiguration in the Enterprise JavaBean Component Model," M.J. Rutherford, K. Anderson, A. Carzaniga, D. Heimbigner, and A.L. Wolf, Proceedings of IFIP/ACM Working Conference on Component Deployment, Berlin, Germany, June 2002.

Copies of the above publications are enclosed with this report.

THE WILLOW SURVIVABILITY ARCHITECTURE

John Knight* Dennis Heimbigner+ Alexander Wolf+

Antonio Carzaniga+ Jonathan Hill* Premkumar Devanbu**

*Department of Computer Science
University of Virginia
Charlottesville, VA 22904-4740
{knight,jch8f}@cs.virginia.edu

+Department of Computer Science
University of Colorado
Boulder, CO 80309-0430
{alw,dennis, carzanig}@cs.colorado.edu

**Department of Computer Science
University of California
Davis, CA 95616-8562
devanbu@cs.ucdavis.edu

Introduction

The Willow architecture provides a comprehensive architectural approach to the provision of survivability[8] in critical information networks. It is based on the notion that survivability of a network requires reconfiguration at both the system and the application levels. The Willow notion of reconfiguration is very general, and the architecture provides reconfiguration mechanisms for both automatic and manual network control.

In this paper we summarize the Willow concepts and provide an overview of the Willow architecture. Finally we describe a demonstration application system that has been built on top of a prototype Willow implementation.

Willow Concepts

The Willow survivability architecture[6] is a secure, automated framework that effects a wide spectrum of both *proactive* and *reactive* reconfigurations of large-scale, heterogeneous, distributed systems. The architecture is designed to enhance the survivability of critical networked information systems by: (a) ensuring that the correct configuration is in place and remains in place during normal operation; (b) facilitating the reconfiguration of such systems in response to anticipated threats before they occur (including security threats); and (c) recovering from damage after it occurs (including security attacks).

Proactive reconfiguration adds, removes, and replaces components and interconnections, or changes their mode of operation. This form of reconfiguration, referred to as *posturing,* is designed to limit possible vulnerabilities when the possibility of a threat that will exploit them is heightened. An example of posturing would be a network-wide shutdown of non-essential services, strengthening of cryptographic keys, and disconnection of non-essential network links if the release of a new worm is expected or infections have already been observed.

In a complementary fashion, reactive reconfiguration adds, removes, and replaces components and interconnections to restore the integrity of a system in bounded time once damage or intrusions have taken place. In Willow, this is, in fact, network *fault tolerance* and mechanisms are provided for both the detection of errors and recovery from them[7]. As an example of fault tolerance, the network might detect a coordinated attack on a distributed application and respond automatically by activating copies of the application modules on different network nodes while configuring the system to ignore the suspect modules. The system would perform this modification rapidly and inform system administrators of the change.

The Willow concept derives from a realization that software configuration control and network fault tolerance are two different aspects of the general problem of overall control of distributed systems. Both utilize specialized knowledge about the applications, the resources available and the network state to prepare and react to changing conditions for applications in a network. The difference lies in the time frames at which the two aspects operate, and in the mechanisms used to detect and respond to circumstances needing action. Network fault tolerance is mostly time-bounded, needing prescribed responses to anticipated faults. Software configuration management involves run-time analysis of network state to determine necessary basic actions from a series of prescribed facts and newly available network state (new software versions, operating system conditions, etc.)

The Willow architecture supports reconfiguration in a very broad sense, and reconfiguration in this context refers to any scenario that is outside of normal, "steady-state" operation. Thus, for example, initial system deployment is included intentionally in this definition as are system modifications, posturing and so on. All are viewed merely as special cases of the general notion of reconfiguration. More specifically, the system reconfigurations supported by Willow are:

- Initial application system deployment.
- Periodic application and operating system updates including component replacement and re-parameterization.
- Planned posture changes in response to anticipated threats.
- Planned fault tolerance in response to anticipated component failures.
- Systematic best efforts to deal with unanticipated failures.

Reconfiguration takes place after a decision is made that it is required. An important element of the Willow approach is the integration of information from sensing mechanisms within the network (such as intrusion detection systems) and information from other sources (such as intelligence data). Since reconfiguration could be used as a means of network attack, the input that is used in the Willow decision-making process is managed by a comprehensive trust mechanism [2].

Summary of the Architecture

The fundamental structure of the Willow architecture is a *control mechanism* that is there to deal with network changes. The individual schemes for dealing with different reconfiguration scenarios might be different, but conceptually they are instances of a common control paradigm that pervades the architecture. This common control loop, and the primary elements of the architecture, are depicted in Figure 1. The control loop contains sensing, diagnosis, synthesis, coordination, and actuation components to affect desired network maintenance.

Fundamental to the implementation of the control loop are: (a) the use of formal languages for the specification of control; (b) large-scale, wide-area communication via the publish-subscribe paradigm; (c) reconfiguration coordination capability; and (d) homogeneous actuation across heterogeneous environments.

The control loop of Figure 1 begins with *sensing* of network state. Sensors can include reports from applications, application heartbeat monitors, intrusion detection alarms, or any other means

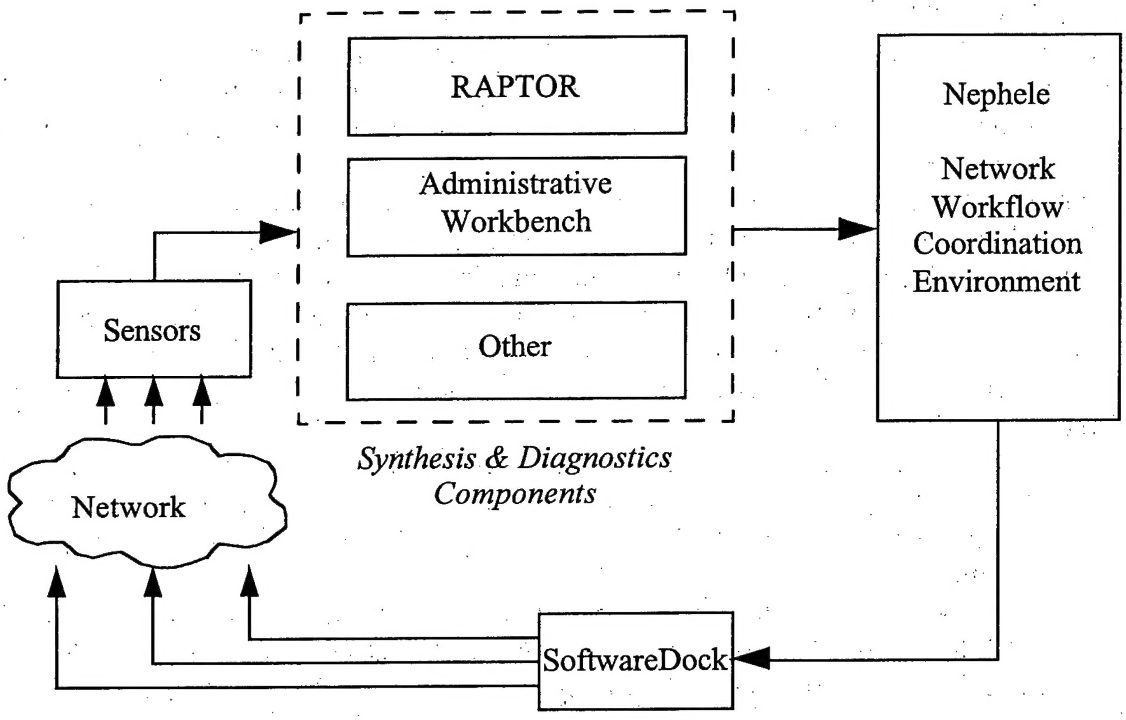

RAPTOR
Administrative Workbench
Other
Synthesis & Diagnostics Components
Nephele
Network Workflow Coordination Environment
Sensors
Network
SoftwareDock

Fig. 1. The Willow architecture.

of measuring actual network properties. From sensing events, independent *diagnosis and synthesis* components build models of network state and determine required network state changes. RAPTOR is a formal-specification driven diagnosis and synthesis system that receives sensor events to analyze and respond with desired network changes, automatically and in bounded time. The Administrative Workbench is an interactive application allowing system administrators to remotely monitor application conditions and adjust application properties. Additional diagnosis and synthesis components can be added by modification of the Willow specification input.

Synthesis components issue their intended network changes as *workflow* requests. Nephele is a large-scale network workflow execution environment. It oversees coordination and arbitrates resource usage between independently synthesized work requests. Different workflows with differing intentions from different diagnosis and synthesis components might conflict, and Nephele maintains ordering of their operation to best meet the survivability goals of the application domain. When workflows are allowed to activate, workflow events are received by the Software Dock and result in local system state changes. The Software Dock infrastructure provides a single interface for *universal actuation* at application nodes across enterprise level networks[3, 4, 5]. Actuation completes the control loop cycle.

All of the components of the Willow architecture interact via the Siena *publish-subscribe* communication system[1]. This allows efficient, scalable event-driven communication to Willow components throughout large-scale networks. In turn, the components of Willow provide efficient, scalable, well-defined, proactive and reactive network change capabilities. This enhances network application survivability, security, and manageability.

Summary of a Case Study

A prototype Willow system has been developed that implements all the different aspects of the architecture mentioned above. As part of an on-going feasibility study, this prototype implementation has been used to execute a prototype implementation of the *Joint Battlespace Infosphere* (JBI) concept.

Instantiations of the JBI concept, once fully developed, will provide advanced information systems for military use. At the heart of the JBI concept is the notion of publish/subscribe semantics in which different information sources publish their data to a network and those interested in the information subscribe to those parts which they wish to see. The expectation is that very large amounst of military information will be published to a JBI and large numbers of consumers (commanders at all levels) will tailor the information they receive by subscribing appropriately.

Clearly a JBI will be an attractive target to an adversary for many reasons. Such a system might be attacked in various ways by hackers or disabled by battle damage, physical terrorism, software faults, and so on. It is essential, therefore, that a JBI be survivable, and the Willow architecture is a candidate implementation platform.

We have developed a JBI implementation based on the Siena publish/subscribe system that includes several information-processing modules (known as fuselets) and synthetic publishers and subscribers. All of the components of the system have been enhanced to allow them to respond appropriately to reconfigration actions. In addition, the different elements of the implementation have been extended deliberately with vulnerabilities so as to permit demonstration and evaluation of the reconfiguration capabilities of Willow.

The JBI implementation operating on the Willow architecture has been subjected to a preliminary evaluation by fault injection. The JBI system has been deployed to a test network entirely automatically and shown to adopt new postures under operator control as desired. The system has also been shown to reconfigure automatically when network faults were injected.

References

[1] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf, Design and Evaluation of a Wide-Area Event Notification Service, ACM Transactions on Computer Systems, 19(3):332-383, Aug 2001.

[2] P. Devanbu, M. Gertz, C. Martel, and S. Stubblebine. Authentic Third-Party Data Publication. In Proceedings of the Fourteenth IFIP Working Conference on Database Security, August 2000. To appear.

[3] R.S. Hall, D.M. Heimbigner, A. van der Hoek, and A.L. Wolf. An Architecture for Post-Development Configuration Management in a Wide-Area Network. In Proceedings of the 1997 International Conference on Distributed Computing Systems, pages 269–278. IEEE Computer Soci-ety, May 1997.

[4] R.S. Hall, D.M. Heimbigner, and A.L. Wolf. Evaluating Software Deployment Languages and Schema. In Proceedings of the 1998 International Conference on Software Maintenance, pages 177– 185. IEEE Computer Society, November 1998.

[5] R.S. Hall, D.M. Heimbigner, and A.L. Wolf. A Cooperative Approach to Support Software Deployment Using the Software Dock. In Proceedings of the 1999 International Conference on Software Engineering, pages 174–183. Association for Computer Machinery, May 1999.

[6] J.C. Knight, K. Sullivan, M.C. Elder, and C. Wang. Survivability Architectures: Issues and Approaches. In Proceedings of the DARPA Information Survivability Conference and Exposition, pages 157–171, Los Alamitos, California, January 2000. IEEE Computer Society Press.

[7] J.C. Knight, M. C. Elder, Fault Tolerant Distributed Information Systems, International Symposium on Software Reliability Engineering, Hong Kong (November 2001).

[8] J.C. Knight and K.J. Sullivan, On the Definition of Survivability, University of Virginia, Department of Computer Science, Technical Report CS-TR-33-00

A Testbed for Configuration Management Policy Programming

André van der Hoek, *Member, IEEE*, Antonio Carzaniga,
Dennis Heimbigner, *Member, IEEE*, and Alexander L. Wolf, *Member, IEEE Computer Society*

Abstract—Even though the number and variety of available configuration management systems has grown rapidly in the past few years, the need for new configuration management systems still remains. Driving this need are the emergence of situations requiring highly specialized solutions, the demand for management of artifacts other than traditional source code and the exploration of entirely new research questions in configuration management. Complicating the picture is the trend toward organizational structures that involve personnel working at physically separate sites. We have developed a testbed to support the rapid development of configuration management systems. The testbed separates configuration management repositories (i.e., the stores for versions of artifacts) from configuration management policies (i.e., the procedures, according to which the versions are manipulated) by providing a generic model of a distributed repository and an associated programmatic interface. Specific configuration management policies are programmed as unique extensions to the generic interface, while the underlying distributed repository is reused across different policies. In this paper, we describe the repository model and its interface and present our experience in using a prototype of the testbed, called NUCM, to implement a variety of configuration management systems.

Index Terms—Configuration management, configuration management policies, distributed configuration management, policy programming, peer-to-peer, version control.

1 Introduction

Since its beginnings in the early 1970s, the field of configuration management (CM) has slowly but surely evolved. The marketplace for CM products is now worth well over one billion dollars per year [8]. More than one hundred commercial CM systems, representing a wide range of functionality, are currently available. While some are simple clones of SCCS [41] and RCS [47], others have pushed the state of the art quite considerably by offering a full spectrum of functionality [14]. Most CM systems, however, fall somewhere in between, each providing some distinguishing combination of functionality.

Despite the variety of available systems, several compelling reasons exist to continue the development of new CM systems. First, in the current generation, the basic functionality provided by a given CM system is fixed; if specialized functionality is needed in a particular situation (e.g., required compliance with company-wide standards [40] or e-mail-based synchronization of workspaces [33]), it becomes difficult to provide. A second reason is that existing CM systems tend to focus on the management of source code; if other types of artifacts need to be managed and configured (e.g., Web sites [34], software architectures [11], or legal databases [29]), only a limited amount of support is available. A third reason is that existing CM systems are based on certain underlying assumptions; if new approaches are developed that are in conflict with some of these assumptions (e.g., the approach based on feature logic [53]), little help is available to implement them.

Already a daunting task in and of itself, the construction of a CM system is further complicated by the fact that many of today's projects are carried out in a distributed fashion. In these projects, multiple collaborating participants are physically dispersed over a number of geographical locations, sometimes even belonging to different companies. Not only does this influence the implementation of a CM system in that it must operate in the context of a wide-area network, it also influences the basic design of a CM system in that its built-in processes must be supportive of distributed and probably decentralized collaboration.

We have developed a testbed to support the rapid construction of new, potentially distributed, CM systems. The testbed embodies an architecture that separates CM *repositories*, which are the stores for versions of software artifacts and information about those artifacts, from CM *policies*, which are the specific procedures for creating, evolving, and assembling versions of artifacts maintained in the repositories. Key to this architecture is the definition of an *abstraction layer* that consists of a generic model of a distributed CM repository and a programmatic interface for implementing, on top of the repository, specific CM policies.

The generic model consists of five components covering the major aspects of a configuration management repository, namely, storage, distribution, naming, access, and attributes. Similarly, the programmatic interface consists of

- *A. van der Hoek is with the Institute for Software Research, Department of Information and Computer Science, University of California, Irvine, CA 92697-3425. E-mail: andre@ics.uci.edu.*
- *A. Carzaniga, D. Heimbigner, and A.L. Wolf are with the Software Engineering Research Laboratory, Department of Computer Science, University of Colorado, Boulder, CO 80309. E-mail: {carzanig,dennis,alw}@cs.colorado.edu.*

Manuscript received 22 Sept. 1998; revised 28 Mar. 2000; accepted 17 Jan. 2001.
Recommended for acceptance by W. Griswold.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 107440.

seven orthogonal categories of functions, including access, versioning, querying, and distribution. CM policies are programmed as extensions to the generic interface, while the underlying distributed repository is reused across different policies. Structured this way, the testbed supports direct and flexible experimentation with new CM policies.

Overall, the design of the abstraction layer was guided by the following high-level objectives:

- *The abstraction layer should be policy independent.* In order for the abstraction layer to support the construction of a wide variety of CM policies, the repository model and programmatic interface should not themselves contain any restrictive policy decisions. For example, if the repository model only provided a facility to store versions of artifacts as a traditional version tree, it would be very difficult to implement the more advanced change-set policy [35]. Similarly, if the functions in the programmatic interface automatically created a new version of an artifact whenever one of its constituent parts is modified, CM policies in which the evolution of an artifact is explicitly managed by a user would, once again, be difficult—if not impossible—to implement.
- *The abstraction layer should support distributed operation.* As proven by the considerable amount of research on the issue [2], [10], [13], [25], providing proper support for the distributed operation of a CM system is a complicated task. Therefore, it is desirable to incorporate support for distribution as an intrinsic property of the repository model and programmatic interface. In particular, the repository model should be able to support a variety of distribution mechanisms (such as peer-to-peer or master-slave) and the programmatic interface should permit the control of the physical placement of artifacts.

  It is important, however, that support for distribution be isolated from other facets of the abstraction. In particular, the low-level details of the distribution aspects of building a CM system (e.g., connection protocols, communication protocols, and time outs) should be isolated from the policy programming aspects by placing those details within the implementation of the repository model. Further, the distribution aspects of relevance to a CM policy (e.g., access to remote repositories and placement of artifacts) should be isolated from the versioning, querying, and other functional categories of CM policy programming. More specifically, the functions in the interface should appear the same regardless of the physical location of the artifacts they manipulate.
- *The abstraction layer should support the management of arbitrary kinds of artifacts.* As previously mentioned, CM systems are increasingly needed to manage artifacts other than source code. To allow such specialized CM systems to be constructed, neither the repository model nor the programmatic interface should make assumptions about the kinds of artifacts that are being manipulated. For example, it is well known that certain algorithms for computing the difference between two versions of an artifact work better for textual data, such as documents and program code, than for binary data, such as images or program executables [27]. Incorporating such a biased differencing algorithm into the abstraction would violate its ability to properly handle different kinds of artifacts.
- *The abstraction layer should be able to support traditional CM functionality.* Even though the abstraction layer is meant to support the construction of new CM policies, it should be obvious that it also must be able to support the construction of existing CM policies. If it could not support the latter, the architectural separation of CM repositories from CM policies results in a loss of functionality and it would be likely that certain variants of existing CM policies could not be implemented.

NUCM (Network-Unified Configuration Management) is our prototype implementation of the testbed. It has been key to the development of several innovative CM systems, including DVS [9] and SRM [49]. As a prototype, NUCM was not designed to exhibit the robustness or completeness that one would expect of a commercial implementation of the abstraction layer. Similarly, the CM systems we built using NUCM were not designed to be particularly robust or complete (although two of them are currently in everyday use). Instead, our focus was on being able to evaluate the utility of the abstraction layer in supporting CM policy programming.

Fig. 1 illustrates the architecture of NUCM in terms of an example repository structure. A CM system that uses a NUCM repository consists of two parts: the generic NUCM client and a particular CM policy. The generic NUCM client implements the programmatic interface and, thus, is the foundation upon which particular CM policies are implemented. This is illustrated in Fig. 1, where two CM policies, namely, policy X and policy Y, both use the generic NUCM client to store and version the artifacts that they manage. In general, a single repository can store artifacts that are managed by different CM policies, as long as the policies partition the artifacts in separate name spaces within the repository. If different policies operate on the same artifacts, it is the responsibility of the CM policies to resolve any conflicts.

The figure also shows that NUCM provides the concept of a *logical* repository that is made up of *physical* repositories. The artifacts in each physical repository are managed by a NUCM *server*. Combined, the NUCM servers for the physical repositories provide access to the complete logical repository. In particular, when artifacts that reside in a different physical repository than the one managed by one of the NUCM servers are requested, that NUCM server will communicate with the other ones to provide access to the artifact.

This paper presents the design of our abstraction layer for CM policy programming and our experiences to date in using NUCM to evaluate the utility of the abstraction layer. We begin in Section 2 by discussing the generic repository model. Section 3 presents the programmatic interface

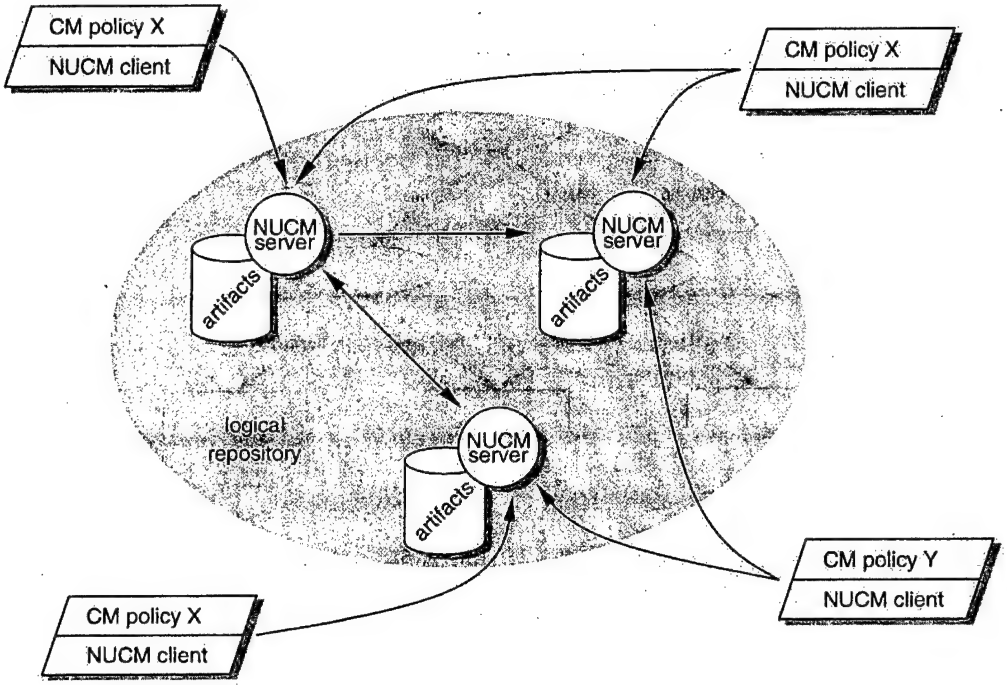


Fig. 1. NUCM architecture.

through which the artifacts are stored and manipulated. Section 4 describes how we have used NUCM to construct a number of rather different CM systems. Related work is discussed in Section 5 and we conclude with a brief look at future work in Section 6.

2 A Generic CM Repository Model

The first part of our abstraction layer is the generic repository model. It consists of five components: a storage model, a distribution model, a naming model, an access model, and an attribute model. The storage model defines the mechanisms for versioning and grouping artifacts, the distribution model defines the different ways in which artifacts can be arranged across different sites, the naming model defines the way individual artifacts can be identified in a distributed repository, the access model defines the primary method of access to artifacts stored in a distributed repository, and the attribute model defines how attributes can be used to associate metadata with artifacts.

A key characteristic of the generic repository model is that, even though specifically designed to support the versioning, grouping, distribution, and other aspects of artifacts, it does not enforce any particular policy for doing so. For instance, while the repository model provides the capability of storing multiple versions of an artifact, it does not impose any specific relationships among those versions. Similarly, the repository model facilitates the storage of different artifacts in different repositories, but it does not enforce a particular organization of the artifacts among the different repositories. In both these and other cases of separation of CM repository from CM policy, it is up to the CM policy programmer to use the interface functions discussed in Section 3 to manipulate the CM repository into the desired behavior.

2.1 Storage Model

The basis for the storage model is a directed graph with two kinds of nodes: *atoms* and *collections*. An atom is a leaf node in a graph and represents a monolithic entity that has no substructure visible to the storage model. Typical atoms include source files or sections of a document. Contrary to atoms, the structure of collections is known to the storage model: Collections are the basic mechanism used to group atoms into named sets. For example, a collection might represent a program that consists of a set of source files. Alternatively, a collection could represent a document that is composed out of a number of sections.

Collections can be used recursively and can themselves be part of larger, higher-level collections. For instance, a collection that represents a system release could consist of a collection for the source code of the system and a collection for the documentation of the system. Membership of a collection can, of course, be mixed: A single collection can contain both atoms and collections. A collection that represents a document could have as its members short sections that are captured as atoms, as well as longer, further subdivided sections that are captured as collections.

Fig. 2 illustrates the basic concepts of atoms, collections, and their member relationships. The figure shows a portion of a repository for the C source code of two hypothetical software systems, `WordProcessor` and `DrawingEditor`. Collections are shown as ovals, atoms as rectangles, and member relationships as arrows. Both software systems not only contain a separate subsystem, as demonstrated by the collections `SpellChecker` and `Menu`, respectively, but they also share a collection called `GUI-lib`. In turn, these lower-level collections simply contain atoms, which, in this example, represent the source files that implement both systems.

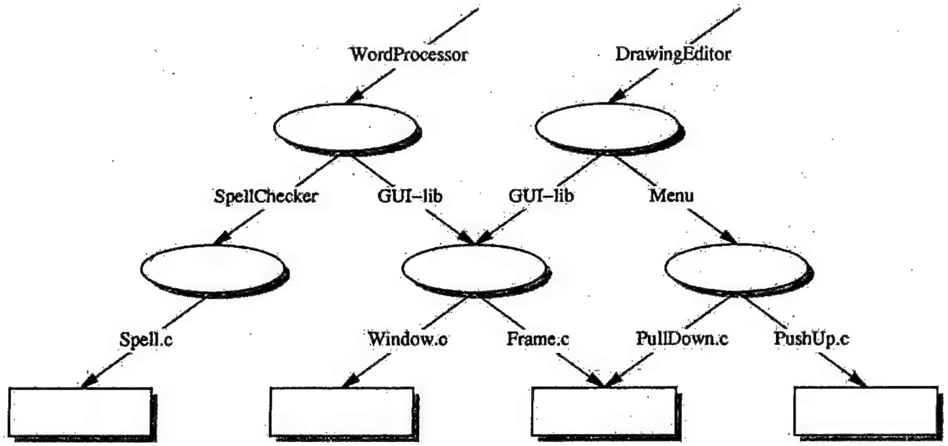


Fig. 2. Example repository contents without versions.

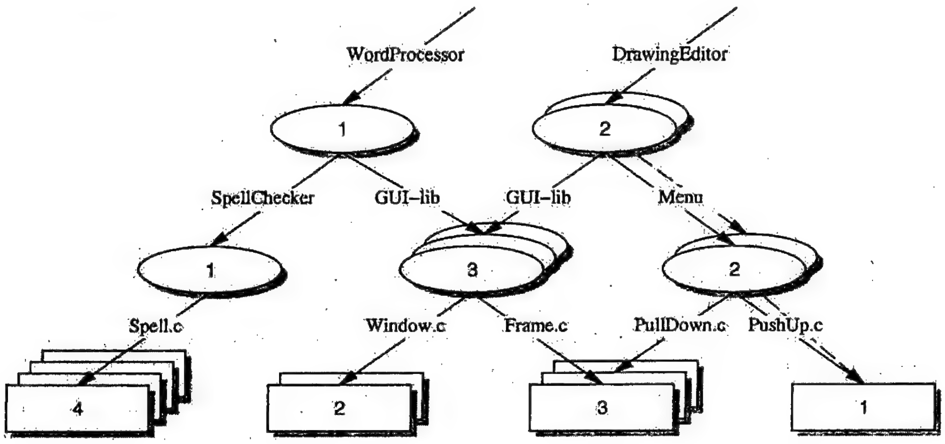


Fig. 3. Example repository contents with versions.

In contrast to other approaches, such as CME [25] or ScmEngine [10], the storage model does not impose any semantic relationship among the versions of an artifact. In particular, the tree-structured revision and variant relationships that are found in many—but by no means all—CM systems are not present in the directed versioned graph. Instead, the graph simply provides a unique number with which to identify each version. This allows a CM system to employ its own type of semantic relationships among versions and, hence, increases the generality of the repository.

The decision not to enforce semantic relationships among the versions of an artifact is based on the more general observation that many such relationships exist. Some examples include *derived-from, is-composed-of, is-part-of, depends-on,* and *includes*. Different CM systems support different subsets of these and other relationships. Therefore, rather than directly maintaining only an arbitrary subset of relationships, the storage model is generic in that it facilitates the creation and maintenance of arbitrary, policy-programmed relationships. It does so through the use of collections to group artifacts and the use of attributes to label versions of artifacts. While this may at first seem inconvenient, since a policy programmer is now expected to implement relationships, the ability to reuse these implementations mitigates the inconvenience. For example, the policy code that defines the version-tree relationship of the WebDAV example in Section 4 reuses much of the policy code of an earlier CM system. This earlier CM system also uses the version-tree relationship and was built using NUCM [50].

Fig. 3 shows how the directed graph of artifacts presented in Fig. 2 is enhanced with versions to form a versioned directed graph. Stacks of ovals and rectangles represent sets of versions of collections and atoms, respectively. Numbers indicate the relative age of versions: the higher the number, the younger the version.[1] Dashed arrows represent the member relationships of older versions of collections. Observe that membership of collections is on a per-version basis. For example, both version 1 and version 2 of the collection `Menu` contain version 1 of the atom `PushUp.c`, but version 2 contains an additional atom, namely, version 2 of the atom `PullDown.c`.

2.2 Distribution Model

The distribution model of the abstraction layer complements the functionality of the storage model. Whereas the storage model specifies how artifacts can be grouped, versioned, and related through the versioned directed graph, the distribution model precisely defines how the versioned directed graph can be stored in a distributed

[1] As further discussed in Section 3, the "age" of versions merely indicates their relative order of creation. In fact, the contents of the versions may, depending on the policy, change over time.

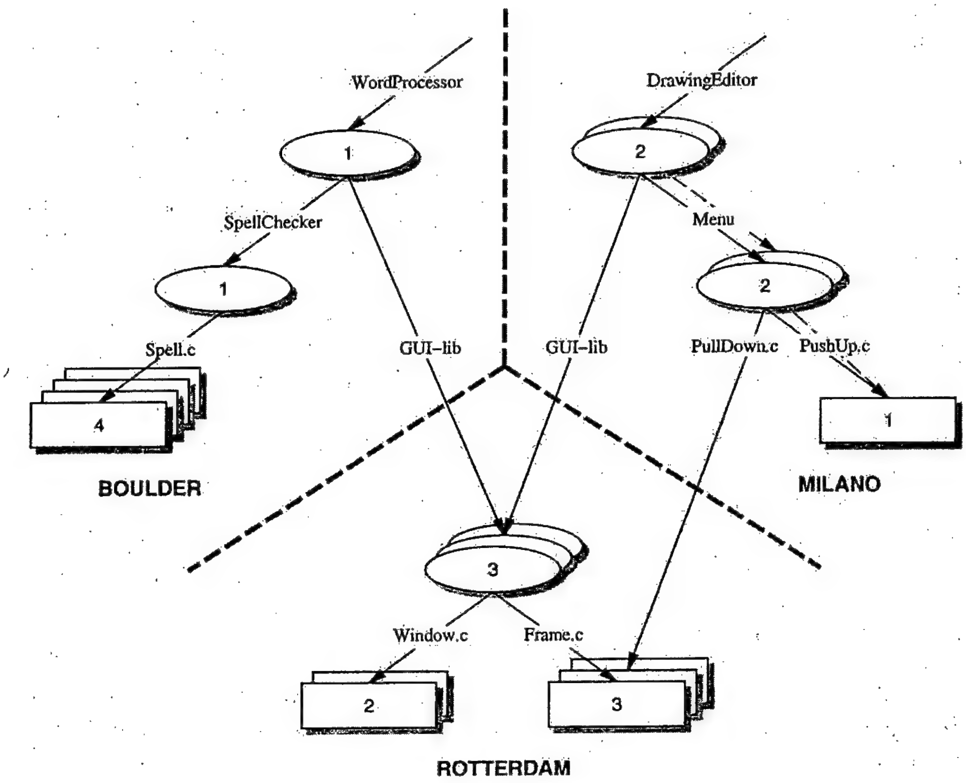


Fig. 4. Example repository contents of Fig. 3 as distributed over three different sites.

fashion. In particular, the distribution model defines two types of repositories: a *physical* repository and a *logical* repository. A physical repository is the actual store for some part of a versioned directed graph at a particular site. It contains, for a number of artifacts, the contents of the versions.

A logical repository is a group of one or more physical repositories that together store a complete versioned directed graph of artifacts. Because the distribution model is policy independent, a requirement for a logical repository is that it has to be able to support the modeling of a variety of distributed CM policies. To do so, physical repositories that are part of a logical repository collaborate in a *peer-to-peer* fashion, with no centralized "master" repository controlling the distribution of artifacts. Instead, distribution is controlled at the individual artifact level: Collections not only maintain the names of their member artifacts, they also track the physical repository in which each member artifact is stored. Thus, member relationships may span the geographical boundaries that exist among physical repositories.

The physical location of artifacts in a logical repository is irrelevant. Artifacts can be obtained from any physical repository that is part of the logical repository, whether the physical repository resides on a local disk, on the local network, or on the other side of the world. Based on the fact that collections keep track of the physical repositories in which their member artifacts reside, requests for member artifacts that are stored at a different physical repository than that of the collection are forwarded. Thus, physical repositories act as both clients and servers, requesting services from each other and fulfilling service requests for each other.

Fig. 4 presents these concepts with an example distributed repository. Shown is the repository of Fig. 3 as distributed over three different sites, namely, Boulder, Milano, and Rotterdam. Each of these sites maintains a physical repository with artifacts. The physical repository located in Boulder maintains the collection `WordProcessor`, the physical repository in Milano maintains the collection `DrawingEditor`, and the physical repository in Rotterdam maintains the collection `GUI-lib`. Because the projects in Boulder and Milano rely on the use of the collection `GUI-lib`, their physical repositories are connected with the physical repository in Rotterdam. Two logical repositories are formed: The physical repositories in Boulder and Rotterdam combine into one logical repository that presents a complete view of the collection `WordProcessor` and its constituent artifacts and the physical repositories in Milano and Rotterdam combine into one logical repository that manages the complete system `DrawingEditor`.

It is important to note that it is the simple presence of member relationships among artifacts in different physical repositories that creates logical repositories. Without the membership of version 2 of the collection `GUI-lib` within version 1 of the collection `WordProcessor`, for example, the logical repository formed by the physical repositories in Boulder and Rotterdam would not exist. Instead, the physical repository of Boulder would be a logical repository all by itself.

The distribution model is versatile: Artifacts can be distributed among physical repositories as desired, a single physical repository can be part of multiple logical repositories, and logical repositories can themselves be part of other logical repositories. This flexibility, combined with a peer-to-peer architecture, allows many different distribution schemes to be mapped onto the distribution model. As further demonstrated elsewhere [48], these schemes include the following:

- a single physical repository that is accessed by many CM clients, thus creating a client-server system like DRCS [36];
- several physical repositories that represent a hierarchy of distributed workspaces in which changes in lower level workspaces are gradually promoted up the hierarchy, thus duplicating the essence of the functionality of such systems as NSE [19] and PCMS [46]; and
- a set of physical repositories that act as replicas, in which the contents of the replicas are periodically synchronized by a merging algorithm, a configuration similar to ClearCase Multisite [2].

These and other approaches to distributed CM can be built using the peer-to-peer architecture. While it is true that a solution based on our generic distribution model might not perform as optimally as a specialized solution for a particular CM policy, the flexibility afforded by the repository model allows experimentation with new distribution policies. Once proven to be of use, the implementation of an experimental policy can be optimized for performance.

2.3 Naming Model

An important issue in distributed systems development is naming. Rather than employing a global naming scheme in which each artifact is assigned a single, unique identifier, the naming model is based on a hierarchical naming scheme. The use of hierarchical naming provides three important advantages. First, it naturally fits the hierarchy that is formed by the directed graph of artifacts as defined by the storage model since each part of a name incrementally indicates which member of a collection is chosen when traversing the directed graph. Second, hierarchical naming provides an advantage of scale by avoiding the need for complicated algorithms that create globally unique identifiers. Lastly, it follows the generally accepted practice of decoupling the name of an artifact from its physical location. In particular, since member relationships can span multiple geographical locations, a hierarchical name simply follows these relations without knowing the actual location of the artifact it designates.

By itself, hierarchical naming is not sufficient. Still open is the choice as to whether each part of a hierarchical name is maintained by an artifact or by its containing collection. To allow a single artifact to exist under different names in different collections (an important facility in current CM systems), the naming model prescribes the latter: Names of artifacts are maintained individually by the collections in which the artifact is a member.

The hierarchical name of an artifact adheres to the following template:

```
//physical-repository/<name[:version]>
  [</name[:version]>...]
```

Thus, names in the abstraction layer can be viewed as URLs that are extended with version qualifiers. Version qualifiers provide a means to specify particular versions of artifacts. Because the storage model allows a single version of an artifact to be a member of multiple (versions of multiple) collections, this kind of naming scheme allows an artifact to have multiple names. For example, assuming the logical repository shown in Fig. 4, the following four names are all equally valid as the fully qualified name of version 1 of the atom `PushUp.c`:

```
//Milano/DrawingEditor:1/Menu:1/PushUp.c:1
//Milano/DrawingEditor:1/Menu:2/PushUp.c:1
//Milano/DrawingEditor:2/Menu:1/PushUp.c:1
//Milano/DrawingEditor:2/Menu:2/PushUp.c:1
```

For convenience, the use of version qualifiers is optional. If a version qualifier is not included, then the interpretation of the name defaults to the version of the artifact that is the actual member of the containing collection. For example, the name

```
//Milano/DrawingEditor:2/Menu/PushUp.c
```

also refers to version 1 of atom `PushUp.c` since version 2 of collection `Menu` is the member of version 2 of collection `DrawingEditor` and version 1 of atom `PushUp.c` is the member of version 2 of collection `Menu`. Thus, it defaults to the following fully qualified name:

```
//Milano/DrawingEditor:2/Menu:2/PushUp.c:1
```

Similarly, because version 2 of collection `DrawingEditor` is the member of the repository in Milano, the following two names also refer to version 1 of atom `PushUp.c`:

```
//Milano/DrawingEditor/Menu:2/PushUp.c
//Milano/DrawingEditor/Menu/PushUp.c
```

2.4 Access Model

The fact that artifacts reside in a logical repository does not necessarily imply that they are directly manipulated there. In fact, it is common practice to build a CM system around the notion of a workspace. A workspace materializes a subset of artifacts in the file system. When designing a CM system, the use of a workspace provides three advantages over direct manipulation. First, it provides an insulated work area in which artifacts can be manipulated without being influenced by the work of others. Second, a workspace provides a form of caching, typically residing much closer in proximity to the originator of changes than the physical repository. Finally, a workspace is unobtrusive in that it provides existing applications with access to versioned artifacts without the need to modify those applications to understand the details of the storage and versioning mechanisms that are used.

For these reasons, the access model prescribes the use of workspaces to access artifacts in a logical repository. Each workspace represents a particular version of a particular collection. The structure of the workspace follows the structure of the file system. In particular, collections materialize as directories, lower-level collections materialize as subdirectories, and atoms materialize as files. For example, version 2 of the collection

DrawingEditor as presented in Fig. 4 would have the following directory structure when materialized into a workspace on a UNIX file system.

```
.../DrawingEditor/GUI-lib/
                           /Menu/PullDown.c
                           /PushUp.c
```

ClearCase [3] manages workspaces in the repository by employing a translucent file system in which operating system calls, such as open, read, and write, are trapped and interpreted by the repository. In contrast, workspaces in our access model follow the model that is used by DRCS [36] and DCVS [24], where materialized artifacts are actual copies in the file system of the artifacts in the repository. The advantage is that proprietary replacements for low-level operating system functions do not have to be created (as with ClearCase) and that less network traffic is incurred.

In traditional CM systems, the user of a workspace is a human. The user of the workspace in our access model, however, is primarily intended to be a CM system that, in turn, provides tailored styles of access to their ultimate human users. This is illustrated in Fig. 5. Three layers, each containing a different representation of the artifacts, can be identified. The bottom layer is the repository that contains all versions of all artifacts. Some of these artifacts will be materialized into a workspace, which is illustrated by the middle layer. The materialized artifacts might be transformed by a CM policy for presentation to a human user, resulting in the top layer. Note that the bottom two layers are standard and managed internally beneath the abstraction layer. The top layer, however, can be of any shape or form, since it is determined by the CM policy program.

2.5 Attribute Model

To facilitate the storage of metadata in a repository, the repository model incorporates a simple attribute model. An attribute in this model is an untyped name/value pair that can be dynamically associated with a particular version of an artifact. Each such version can have its own unique set of attributes, and this set can change over time. The CM policy determines both a naming scheme for the attributes and a set of values that the attributes may assume.

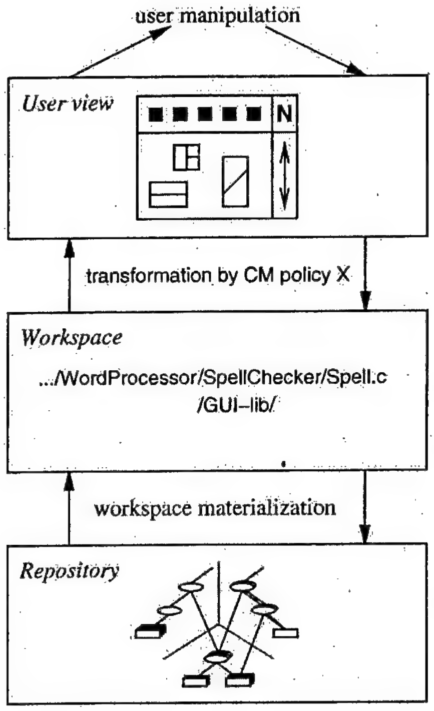
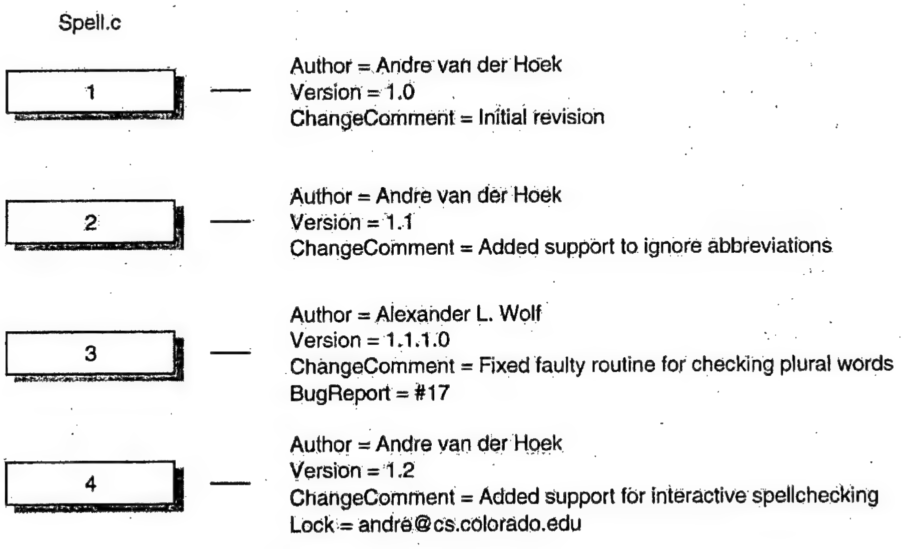

user manipulation
User view
N
transformation by CM policy X
Workspace
.../WordProcessor/SpellChecker/Spell.c
/GUI-lib/
workspace materialization
Repository

Fig. 5. Access model.

An example is provided in Fig. 6, which shows the attributes that have been associated with the various versions of the atom Spell.c. The CM policy managing these versions has labeled them all with the attributes Author, Version, and ChangeComment. Furthermore, if a new version of an artifact fixes a previously identified bug, that version will be labeled with the attribute BugReport, which contains the number of the bug report that describes the bug that is resolved. Finally, if a version of an artifact is locked, the attribute Lock is set



Author = Andre van der Hoek
Version = 1.0
ChangeComment = Initial revision

Author = Andre van der Hoek
Version = 1.1
ChangeComment = Added support to ignore abbreviations

Author = Alexander L. Wolf
Version = 1.1.1.0
ChangeComment = Fixed faulty routine for checking plural words
BugReport = #17

Author = Andre van der Hoek
Version = 1.2
ChangeComment = Added support for interactive spellchecking
Lock = andre@cs.colorado.edu

Fig. 6. Example attributes associated with the atom Spell.c.

to contain the e-mail address of the person who has placed the lock. Note that some attributes contain values that are assigned by the CM policy itself (e.g., Author, Version, and Lock), whereas other attributes contain values that are supplied by users of the CM policy (e.g., BugReport and ChangeComment).

3 Repository Interface

The second component of the abstraction layer defined by the testbed is the programmatic interface through which artifacts that are stored in a repository can be manipulated by CM policies. The complete interface consists of seven categories of functionality. These categories, listed in Table 1 with the functions they contain, are the following: access functions, which provide access to artifacts in a repository by materializing them in a workspace; versioning functions, which manage the way artifacts evolve into new versions; collection functions, which manage the membership of collections; distribution functions, which control the placement of artifacts in specific physical repositories; a deletion function, which allows a CM policy to remove artifacts from a repository; query functions, which provide a CM policy with various kinds of information about the state of artifacts in a repository or workspace; and attribute functions, which manage the association of attributes with versions of artifacts.

A CM policy is built by programming against the interface and using combinations of interface functions to implement the particular functionality needed. Because a wide range of CM policies has to be supported, the interface functions—much like the various submodels in the repository model—do not impose any particular CM policy. Instead, they provide the mechanisms for CM systems to implement specific policies. While the particular semantics of the interface functions might therefore seem odd from the perspective of a human user, those same semantics are invaluable to a CM policy programmer.

An important characteristic of the programmatic interface is the orthogonality among the various functional categories. For example, the distribution functions are the only functions concerned with the distributed nature of a repository. The other functions are not influenced by the fact that artifacts are stored in different locations. Their behavior is the same, regardless of whether the artifacts are managed by a local repository or a remote one. Similarly, the collection functions are the only functions that recognize the special nature of collections. The other functions in the programmatic interface behave the same, irrespective of whether they operate on atoms or collections.

It should be noted that the functionality offered by each individual interface function is rather limited. At first, this seems contradictory to the goal of providing a high-level interface for configuration management policy programming. However, because of the limited functionality, each function can be defined with precise semantics. Not only does that generalize the applicability of the interface functions, it also allows the rapid construction of particular CM policies through the composition of sets of interface functions. In Section 4, we present some of the CM policies that we have constructed this way. Below, we introduce, per category, the individual interface functions that constitute the programmatic interface to the generic repository model.

3.1 Access Functions

Access to the artifacts in a repository is, as discussed in Section 2.4, obtained through a workspace in which artifacts are materialized upon request. Once the artifacts are materialized, other interface functions become available to manipulate them. In particular, versioning functions can be used to store new instances of artifacts, and collection functions can be used to manipulate the membership of collections.

The access functions in the programmatic interface are nc_open and nc_close. The function nc_open provides access to a particular version of an artifact by materializing it in a workspace. Atoms are materialized as files, collections as directories. Each use of the function nc_open materializes a single artifact. A workspace, then, has to be constructed in an incremental fashion. This mechanism allows a CM system to populate a workspace with only the artifacts that it needs. The function nc_close is used to remove artifacts from a workspace. The function operates in a recursive manner: When a collection is closed, all the artifacts that it contains are removed from the workspace as well.

3.2 Versioning Functions

Once an artifact has been opened in a workspace, the following versioning functions become available to create and store new versions of the artifact:

nc_initiatechange,
nc_abortchange,
nc_commitchange, and
nc_commitchangeandreplace.

Through the function nc_initiatechange, a CM policy informs a workspace of its intention to make a change to an atom or a collection. In response, permission is granted to change the artifact in the workspace. If the artifact is an atom, it can be manipulated by any user program since its contents are not interpreted by the model or interface. A collection, on the other hand, can only be manipulated through the use of collection functions because those functions preserve its special nature (see Section 3.3).

Permission to change an artifact in one workspace does not preclude that artifact from being changed simultaneously in another workspace. In particular, the function nc_initiatechange does not lock an artifact. If a locking protocol is desired, then the attribute functions described in Section 3.7 can be used to construct that protocol. This orthogonality of locking and versioning permits the development of CM policies that range from the optimistic, in which artifacts are not locked and changes are merged when conflicts arise, to the pessimistic, in which artifacts are locked to avoid conflicts.

The function nc_abortchange abandons an intended change to an artifact. It reverts the materialized state of the artifact back to the state that it was in before the function nc_initiatechange was invoked. An nc_abortchange performed on a collection can only succeed if no artifacts that are part of the collection are currently in a state that allows them to be changed. This forces the CM system either

TABLE 1
Programmatic Interface Functions

Category	Function	Description
Access	nc_open	Materializes an artifact version into a workspace.
	nc_close	Removes an artifact version from a workspace.
Versioning	nc_initiatechange	Allows an artifact version in a workspace to be modified.
	nc_abortchange	Returns an artifact version in a workspace to the state it was in before it was initiated for change.
	nc_commitchange	Stores a new version of an artifact in a repository.
	nc_commitchangeandreplace	Overwrites the current version of an artifact in a repository.
Collection	nc_add	Adds an artifact version to a collection.
	nc_remove	Removes an artifact version from a collection.
	nc_rename	Renames an artifact within a collection.
	nc_replaceversion	Replaces the version of an artifact within a collection.
	nc_copy	Copies the versions of an artifact and adds a version of the new artifact to a collection.
	nc_list	Determines the member artifact versions of a collection.
Distribution	nc_setmyserver	Sets the default physical repository in which new artifacts will be stored.
	nc_getlocation	Determines the physical repository that contains the versions of an artifact.
	nc_move	Moves an artifact and its versions from one physical repository to another.
Deletion	nc_destroyversion	Physically removes an artifact version from a repository.
Query	nc_gettype	Determines the kind of an artifact.
	nc_version	Determines the current version of an artifact.
	nc_lastversion	Determines the latest version of an artifact in a repository.
	nc_existsversion	Determines whether a version of an artifact exists in a repository.
	nc_isinitiated	Determines whether an artifact version has been initiated for change in a workspace.
	nc_isopen	Determines whether an artifact version has been materialized into a workspace.
Attribute	nc_testandsetattribute	Associates an attribute and its value with an artifact version (if the attribute does not yet exist).
	nc_setattribute	Associates an attribute and its value with an artifact version (whether or not the attribute exists).
	nc_getattributevalue	Determines the value of an attribute of an artifact version.
	nc_removeattribute	Disassociates an attribute from an artifact version.
	nc_selectversions	Determines the set of versions of an artifact for which an attribute has a certain value.

to commit any changes or to abandon them, thereby avoiding unintentional loss of changes.

To store the changes that have been made to an artifact, two alternative functions can be used. The first, nc_commitchange, commits the changes by storing a new version of the artifact in the repository. It is the only function in the programmatic interface that actually creates new versions of artifacts. None of the other functions have this capability, neither directly nor as a side effect. The second function used to store changes to

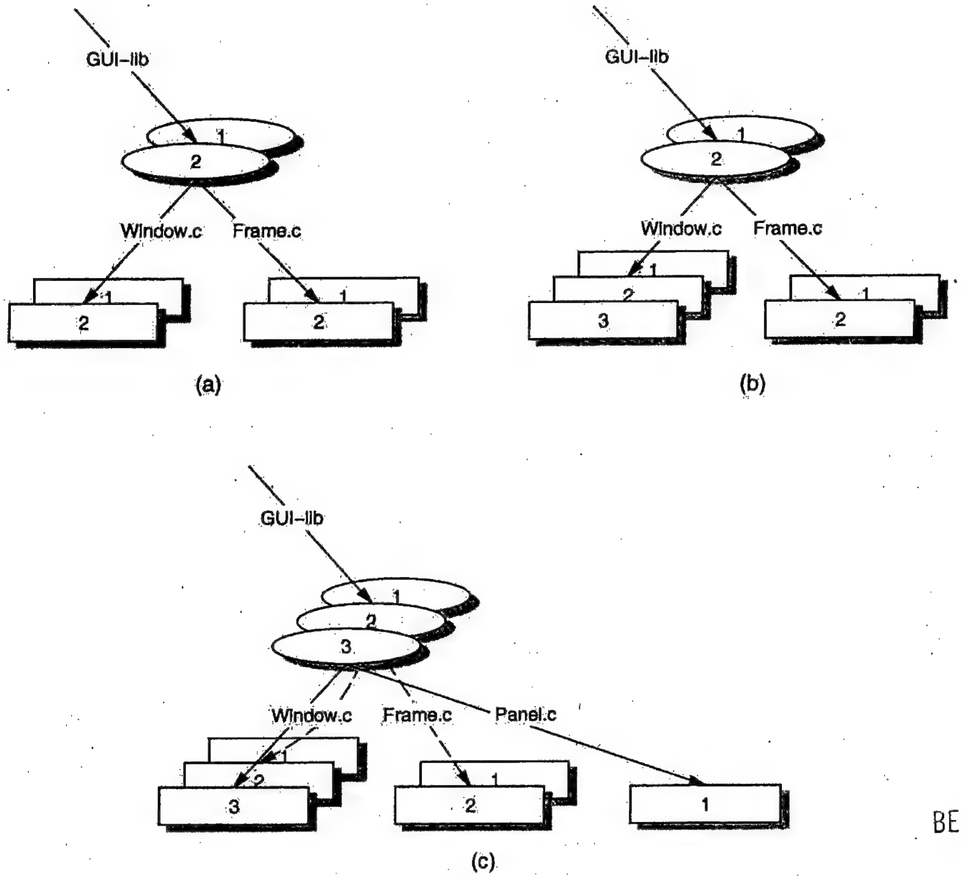


Fig. 7. Progressive states of an example collection.

artifacts is nc_commitchangeandreplace. As its name implies, this function is similar in behavior to the function nc_commitchange, but instead of creating a new version of the artifact, it overwrites the contents of the version that was initiated for change. Both functions, in addition to storing the new contents of the artifact in the repository, also revoke the permission to make further changes to the artifact in the workspace. Once again, locking is an orthogonal concern that is managed with a different category of functions. Therefore, neither function releases any locks that may be held.

The availability of these alternative storage functions allows a CM policy programmer to choose whether particular changes lead to new versions of artifacts or not. This is an especially important decision in the case of collections. Whereas some CM policies prescribe that any change to a member artifact leads to a new version of the collection (e.g., Poem [30] or CoED [6]), other CM policies only version collections when the actual structure of the collection (i.e., its artifact membership) has changed (e.g., ShapeTools [31] or ClearCase [3]). Since this is a policy decision, the programmatic interface facilitates both cases. To model the first case, the function nc_commitchange is used on the collection, whereas the latter case requires the use of the function nc_commitchangeandreplace. Given that an artifact can be a member of multiple collections, a CM policy could even choose to use a different approach for each collection.

To illustrate the versioning functions, suppose we have a repository containing the artifacts depicted in Fig. 7a. Assume further that, using the function nc_open, a workspace has been created that contains version 2 of the artifact GUI-lib and its containing artifacts. To be able to modify the atom Window.c in the workspace, we invoke the function nc_initiatechange. Once the desired changes have been made, we use the function nc_commitchange to store a new version of the atom Window.c in the repository. The result is shown in Fig. 7b. The repository now contains three versions of the atom Window.c, but note that the collection GUI-lib has not changed since we did not invoke the function nc_initiatechange on that collection. Had we used the function nc_commitchangeandreplace instead of the function nc_commitchange, no new version would have been created for the artifact Window.c. In fact, the structure of the repository would still have looked like the one of Fig. 7a, even though the actual contents of version 2 of the atom Window.c would have changed.

3.3 Collection Functions

Similar to the way an editor can be used to change an atom in a workspace, collections need to be changed via some kind of mechanism. Because collections have special semantics, it would be unwise to allow them to be edited directly. Therefore, the programmatic interface contains a number of functions that preserve the semantics of collections while updating their contents. These functions are the following: nc_add, nc_remove, nc_rename, nc_replaceversion, nc_copy, and nc_list. An important aspect of these functions is that they do not directly modify collections in the repository. Instead, they can only modify collections that have been materialized (and initiated for change) in a workspace. To promote these changes to the repository, the versioning functions described in the previous section must be used. This scheme allows many changes to a single collection to be grouped into a single change in the repository.

The functions nc_add and nc_remove behave as expected, adding and removing a version of an artifact to and from a collection, respectively. The function nc_add can add either a new or an existing artifact to a collection. The addition of a new artifact will simply store its contents in the repository. The addition of an existing artifact, on the other hand, will result in an artifact that is shared by multiple collections and for which a single version history is maintained (such as the collection GUI-lib in Fig. 4). If, instead of a shared version history, a separate version history is desired, then the function nc_copy must be used in place of the function nc_add. A distinctly new artifact will be created in the repository. The new artifact will contain the same version history as the artifact that was copied, but the new artifact evolves separately.

A feature that has been difficult to provide in CM systems is the ability to rename artifacts. The testbed solves this problem by providing, directly in its programmatic interface, the function nc_rename. Because an artifact is only renamed within a single collection at a time, it is possible for an artifact to exist under different names in different collections. This is an important feature of the programmatic interface since it allows an artifact to evolve without compromising its naming history.

The function nc_replaceversion complements the other collection functions because it operates in the version dimension as opposed to the naming dimension. Its behavior is simple: It changes the member version of an artifact in a collection to another version. A situation for which this functionality is especially useful is when a CM policy programmer would like to provide an "undo" facility. For example, the facility can be used to replace a newer version of an artifact in a collection with an older one.

The function nc_list rounds out the collection functions. It returns a list of the names and versions of the artifacts that are members of a collection. This functionality is useful in building a CM system that, for example, presents a user with the differences between two versions of a collection, recursively opens a workspace, or simply allows a user to dynamically select which artifacts to lock or check out.

The set of collection functions is complete. If we consider the artifacts that are members of a collection to be organized in a two-dimensional space defined by name and version, all primitive functionality is provided. A name-version pair can be added, a name-version pair can be removed, a name is allowed to change, and a version is allowed to change. Despite the rather primitive functionality provided by each individual function, the complete set of collection functions allows for the rapid construction of higher-level, more powerful functions. For example, a function that replaces, under the same name, one atom with another, can be constructed as a sequence of nc_remove, nc_add, and nc_rename.

To illustrate the collection functions, we continue the example of Fig. 7b. Assume that all artifacts are still open in the workspace. To manipulate the collection GUI-lib, the function nc_initiatechange is first used to gain proper permission. Then, to update the atom Window.c to its latest version, the function nc_replaceversion is applied. In addition, to provide a panel as opposed to a frame in the collection GUI-lib, the function nc_remove is used to remove the atom Frame.c and the function nc_add is used to add the atom Panel.c. These changes are transferred to the repository using the function nc_commitchange. As a result, the repository looks as shown in Fig. 7c. A new version of the collection GUI-lib has been created that reflects the new membership. In addition, because we used the function nc_commitchange instead of the function nc_cimmitchangeandreplace, the old version of the collection is still available. This means that, if the function nc_list is used on version 2 of the collection GUI-lib, then version 2 of the atom Window.c and version 2 of the atom Frame.c are listed as the collection members, whereas, if the function nc_list is used on version 3 of the collection, then version 3 of atom Window.c and version 1 of atom Panel.c are listed as members.

3.4 Distribution Functions

An important aspect of the distribution model discussed in Section 2.2 is that it isolates distribution. This is reflected in the semantics of the various interface functions since the functions behave the same whether artifacts are stored locally or remotely. On the other hand, sometimes a need exists for control over the location of artifacts. Users of systems that completely hide distribution often encounter performance difficulties related to the physical placement of data. To counter this problem, the programmatic interface contains functions that allow a CM system to determine and change the physical location of artifacts within a logical repository.

The first function, nc_setmyserver, specifies the default physical repository to which newly created artifacts are added. New artifacts can be added to any physical repository since it is not required that they are added to the same physical repository as the one in which their parent collection resides. When a new artifact is added to a different repository, a connection is made between that repository and the repository in which the parent collection is located. This connection is the bridge that forms the logical repository spanning the two physical repositories.

To determine the actual location of an artifact, the function nc_getlocation is used. It returns the physical repository in which an artifact is stored. This information can, in turn, be used by the function nc_move to collocate artifacts that are regularly used together or to move artifacts to those physical repositories that are closer in proximity to the workspaces in which they are manipulated. To comply with the requirement set forth by the distribution model that all versions of an artifact are located in a single physical repository, the function nc_move moves the complete version history of an artifact from one physical repository to another.

3.5 Deletion Function

Since it violates the basic premise of always having a precise history of all changes to all artifacts, deleting (versions of) artifacts from a repository is an uncommon practice in the domain of configuration management. Nevertheless, it should still be possible to do so. Therefore, the function nc_destroyversion is provided in the programmatic interface to physically delete a particular version of an artifact from a repository. During deletion, however, a specific rule is enforced: A version of an artifact can only be deleted if that version is not a member of a collection. Consider the example in Fig. 7c. A CM policy is allowed to delete version 1 of atom Window.c, but the deletion of version 2 is disallowed because it is a member of version 2 of the collection GUI-lib. While it may seem as though we are making a policy decision through this restriction, it is one that is intended simply to preserve the consistency of the repository structure.

The function nc_destroyversion, by itself, is not sufficient to be able to delete all artifacts from a repository. A second, implicit form of deletion has to be provided by an implementation of the abstraction layer that complements the explicit use of the function nc_destroyversion. The implicit deletion has to take care of two specific cases. First, by allowing artifacts to be removed from a collection with the function nc_remove, it is possible that none of the versions of a certain artifact can be reached in the versioned directed graph of artifacts (consider applying the function nc_remove on version 2 of the atom Frame.c). Second, it is possible to create a sequence in which a new artifact is added to a collection in a workspace with the function nc_add, but removed from that collection by the function nc_remove before a new version of the collection is stored in the repository. In both cases, the storage space occupied by the artifact needs to be automatically reclaimed by an implementation of the abstraction layer.

3.6 Query Functions

The programmatic interface would not be complete without the ability to examine the state of artifacts. For example, when multiple clients share access to an artifact, they should be able to determine whether any new versions of the artifact have been created by another client. The query functions were designed to provide this type of functionality. Although simple, these functions are essential in the development of CM policies because they provide state information that a CM system would otherwise have to determine and track itself. The query functions that provide information about the artifacts in a workspace are particularly important in this respect.

Although the names of the interface functions speak for themselves, we provide a one-sentence description and typical use of each. The function nc_gettype determines whether an artifact is a collection or an atom and is often used when recursively opening a collection and all its containing artifacts in a workspace. To manage version relationships, such as a revision history, the function nc_version can be used to determine the version of an artifact before and after the function nc_commitchange has been used to store some changes. The function nc_lastversion returns the version number of the last version of an artifact, and is used to check for new versions of an artifact that may have been added by another client. If some versions of an artifact have been deleted from a repository, the function nc_existsversion can be used to verify whether or not a particular version is still available. Finally, the functions nc_isopen and nc_isinitiated operate on artifacts in a workspace and are used to verify whether an artifact has been opened and whether it is allowed to change, respectively.

3.7 Attribute Functions

To facilitate, in accordance with the attribute model, the association of metadata with the artifacts in a repository, the programmatic interface contains a number of primitive functions to manipulate attributes. In particular, it is possible to set the value of an attribute with either the function nc_setattribute, which sets the value of an attribute irrespective of whether a value is already set, or the function nc_testandsetattribute, which only sets the value of an attribute when the attribute is currently nonexistent. To remove an attribute, the function nc_removeattribute is used. This function removes both the attribute and its associated value. To search the attributes that may be set on the various versions of an artifact, the function nc_selectversions is used: For a particular artifact in the repository and for a desired attribute value, it returns the version numbers of those versions whose corresponding attribute matches the value.

The attribute functions serve a dual purpose. First, they are used to simply attach metadata to individual versions of an artifact. For example, it is possible to capture such characteristics as the author and creation date of the version, one or more change request identifiers that identify which particular change requests have been incorporated, and a short synopsis of the changes made with respect to the previous version.

The second purpose for which the attribute functions were designed is to support an artifact locking mechanism. In particular, the function nc_testandsetattribute only sets the value of an attribute if it does not yet exist. Therefore, the function can be used to create a lock on an artifact by simply setting an attribute that represents the lock. If the artifact had been previously locked (i.e., the attribute is set), then the function and, hence, the lock attempt will fail. If the attribute had not been previously locked (i.e., the attribute is not set), then the function and lock attempt will succeed. The function nc_removeattribute unlocks the artifact by removing the attribute.

Because of their generic nature, the attribute functions do not themselves enforce locks. Any enforcement results from the usage protocol employed by a CM policy. For example, a lock can be "broken" (intentionally or unintentionally) by using the function nc_setattribute on an existing lock attribute since the function will not fail to set the attribute even though the attribute already exists. In a similar vein, the interpretation of a lock on a collection is left to the CM policy: Does it mean that only the collection itself is locked or does it mean that anything reachable from the collection is also locked? The usage protocol employed by the CM policy will provide an answer that is consistent with the policy it seeks to implement.

Although using the attribute functions for purposes of artifact locking results in a rather primitive mechanism, the functions are powerful enough to directly model the locking schemes employed in such existing CM systems as RCS [47], CCC/Harvest [44], and others. If more sophisticated locking schemes are required, then a separate lock manager, such as Pern [23], should be used instead. This approach is consistent with the desire for locking to be orthogonal to the other functionalities of the interface.

4 Three NUCM-Based CM Systems

The abstraction layer, including its repository model and programmatic interface, has been implemented in the NUCM prototype[2] and was used to develop several CM systems, including the three, rather different ones described in this section. At present, two of those systems, namely, DVS [9] and SRM [49], are in everyday use, while the third system, WebDAV, represents an experimental implementation of an emerging standard in Web versioning [22], [52]. We also created proof-of-concept implementations of the widely known check-in/check-out and change-set policies [18], but used an earlier version of the prototype to do so; those implementations are presented elsewhere [50].

Below, we discuss each system and use parts of their implementations to illustrate how NUCM can be used to program particular CM policies. It should be noted that the policies themselves are not the contribution. Instead, the strength of NUCM lies in the ease with which these policies were constructed and the limited amount of work needed to make them suitable for use in a wide-area setting.

4.1 DVS

DVS (Distributed Versioning System) [9] is a versioning system that is focused on providing a distributed environment in which documents can be authored collaboratively.[3] DVS is centered around the notion of workspaces. Specifically, individual users populate their workspace with the artifacts needed, lock the artifacts they intend to change, modify these artifacts using appropriate tools, and commit their changes from the workspace to a storage facility. This policy is similar to the one employed by RCS [47], except that DVS explicitly recognizes and versions collections and, moreover, operates in a wide-area setting.

DVS exhibits several characteristics that illustrate the power of the abstraction layer.

- *No special code needed to be developed for DVS to operate across a network.* DVS relies entirely on the mechanisms included in NUCM to support distribution. In fact, DVS can not only operate in a client-server mode, but it is also possible to federate multiple physical repositories into a single logical repository that is used by DVS.
- *Only approximately 3,000 new lines of source code were needed to create the full functionality of DVS.*[4] The newly written source code mainly deals with the text-based user interface, the recursive operations on workspaces, the proper locking of artifacts, and the storage of metadata about the artifacts that are versioned. Other functionality, such as distribution, collections, and basic versioning, is inherited from NUCM.
- *The separation of policy from repository allows certain evolutions in the policies to occur incrementally.* This characteristic of NUCM came upon us unexpectedly. On one occasion, DVS was being used by 10 people at five different sites to jointly author a document. It turned out that the policy provided by DVS did not completely match the desired process. In response, some of the DVS functionality was changed and some new functionality was added. When the second version of DVS was subsequently and incrementally deployed to the various sites, no disruption of work occurred. Because the NUCM repository required no downtime and the artifacts in the repository needed no change, slightly different policies could be used by multiple authors at the same time.

To demonstrate how DVS is built upon the functions in the NUCM interface, Fig. 8 presents a portion of the DVS source code. (Note that all error handling has been removed from the example source code shown in this section.) The use of a NUCM interface function is highlighted by a "*". Illustrated is a procedure that synchronizes a workspace with the latest versions of the artifacts in a NUCM repository. The procedure allows either a single artifact or a recursive set of artifacts to be synchronized, depending on the value of the parameter `recursive`.

The procedure consists of three parts. In the first part, the version of the artifact in the workspace and its latest version in the repository are determined through the use of the functions `nc_version` and `nc_lastversion`. If these versions are the same, the artifact is up to date with respect to the repository. If they are not, the second part of the procedure takes care of synchronizing the two by replacing the version in the workspace with the version in the repository. Additionally, before the latest version of the artifact is opened in the workspace, the current version is closed if it had been opened previously. To avoid the loss of changes that may have been made to an artifact, the routine first verifies whether the current version has been initiated for change. If so, the current version of the artifact is preserved in the workspace and the artifact is not synchronized. The third and final part of the procedure deals with the recursive nature of the synchronization of a workspace. If the artifact to be synchronized is a collection, its contained artifacts are obtained and each of these artifacts is in turn synchronized through a recursive call.

4.2 SRM

SRM (Software Release Manager) [43], [49] is a tool that addresses the problem of software release management.[5] SRM supports the release of "systems of systems" from multiple, geographically distributed sites. In particular,

[2] http://www.cs.colorado.edu/serl/cm/nucm.html.

[3] http://www.cs.colorado.edu/serl/cm/dvs.html.

[4] In this paper, all counts of source code lines include empty lines and comments.

[5] http://www.cs.colorado.edu/serl/cm/SRM.html.

```
1      int synchronize_workspace(const char* pathname, int recursive)
2      {
3         //
4         // Part 1:  Determine the current and latest version of the artifact.
5         //
6         strip_version_r(pathname, strippedpath);
7  *      nc_version(strippedpath, "", currentversion);
8  *      nc_lastversion(strippedpath, "", lastversion);
9
10        //
11        // Part 2:  If needed, get the latest version of the artifact, unless
12        // it is checked out.
13        //
14        if (strcmp(lastversion, currentversion) != 0) {
15           do_open = 1;
16 *         if (nc_isopen(strippedpath, "."))
17 *            if (!nc_isinitiated(strippedpath, "."))
18 *               nc_close(strippedpath, ".", 0);
19            else
20               do_open = 0;
21           if (do_open) {
22              set_version_x(strippedpath, lastversion);
23 *            nc_open(strippedpath, ".", ".", "");
24           }
25        }
26
27        //
28        // Part 3:  If necessary, recursively synchronize the workspace.
29        //
30        if (recursive) {
31 *         atype = nc_gettype(strippedpath, ".");
32           if (atype == COLLECTION) {
33 *            nc_list(strippedpath, "", &members);
34              chdir(strippedpath);
35              start = members;
36              while (members != NULL) {
37                 synchronize_workspace(members->name, recursive);
38                 members = members->next;
39              }
40 *            nc_destroy_memberlist(start);
41              if (strcmp(strippedpath, ".") != 0)
42                 chdir("..");
43              }
44           }
45        }
```

BEST AVAILABLE COPY

Fig. 8. DVS routine to synchronize a workspace.

SRM tracks dependency information to automate the packaging and retrieval of components. Software vendors are supported by a simple release process that hides the physical location of dependent components. Customers are supported by a simple retrieval process that allows selection and downloading of components whose physical locations are hidden.

Although SRM is not a traditional CM system that stores and versions source code, it has many similarities to a CM system: It needs to manage multiple releases, it needs to manage dependencies among these releases, and it needs to store metadata about the releases. Combined with the need for a distributed repository that allows multiple sites to collaborate in the release process, these similarities led to the choice of NUCM as the platform upon which to build SRM.

Of relevance to the discussion in this paper is the flexibility that NUCM provides in the creation of a distributed repository. In particular, we examine the way new participants can join a federated SRM repository. To facilitate this functionality, each participating site maintains a NUCM repository that contains the releases they have created. Additionally, one of the NUCM repositories in the federation maintains a collection that contains all releases from all sites. This is illustrated in Fig. 9 by the repositories in Rotterdam and Boulder. Both repositories contain a collection `nucm_root` that contains a local collection `my_releases` and a global collection `all_releases`. In each repository, the collection `my_releases` contains the releases made by that site. The collection `all_releases`, which is shared by both sites, contains all the releases. Note that the repository in Milano is not part of the SRM federation at this point.

The procedure `join`, presented in Fig. 10, illustrates how a new physical repository can join an existing SRM federation. It operates by creating a new collection for local releases, `my_releases`, and linking to the existing collection that contains all releases, `all_releases`. To do so, it first sets up a workspace containing its main collection `nucm_root`, then allows the collection to change

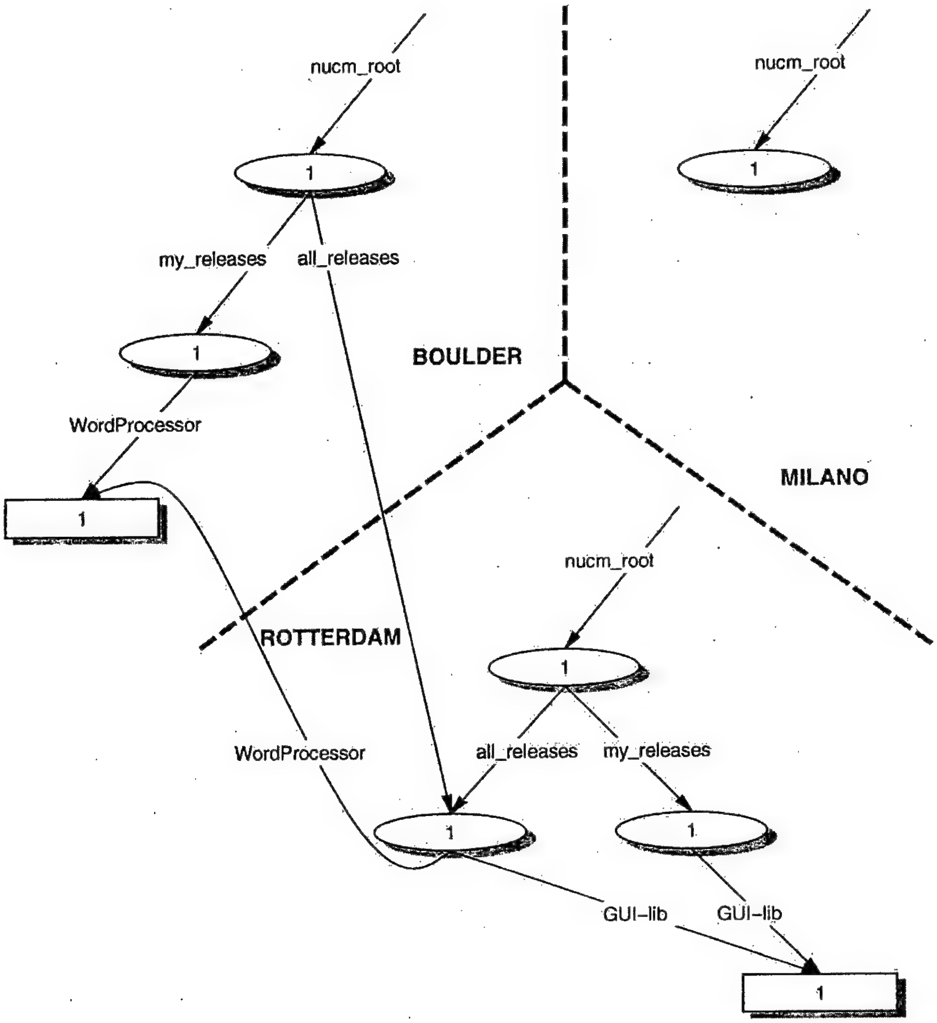


Fig. 9. Federated SRM repository before Milano joins the federation.

```
1     int join(const char* host, const char* port)
2     {
3        //
4        // Part 1: Set up a workspace.
5        //
6        sprintf(all_releases, "//%s:%s/nucm_root/all_releases", host, port);
7        sprintf(my_nucmroot, "//%s:%s/nucm_root", NUCMHOST, NUCMPORT);
8        sprintf(my_releases, "WORKSPACE/my_releases");
9        sprintf(collection, "WORKSPACE/nucm_root");
10 *     nc_open(my_nucmroot, "", WORKSPACE, "");
11 *     nc_initiatechange(collection);
12
13       //
14       // Part 2: Add a new artifact for my personal releases.
15       //
16       mkdir(my_releases);
17 *     nc_add(my_releases, "", collection, "");
18
19       //
20       // Part 3: Import an existing artifact for the list of all releases.
21       //
22 *     nc_add(all_releases, "", collection, "");
23 *     nc_commitchangeandreplace(collection, "");
24 *     nc_close(collection, "", 0);
25    }
```

Fig. 10. SRM routine to join a federation.

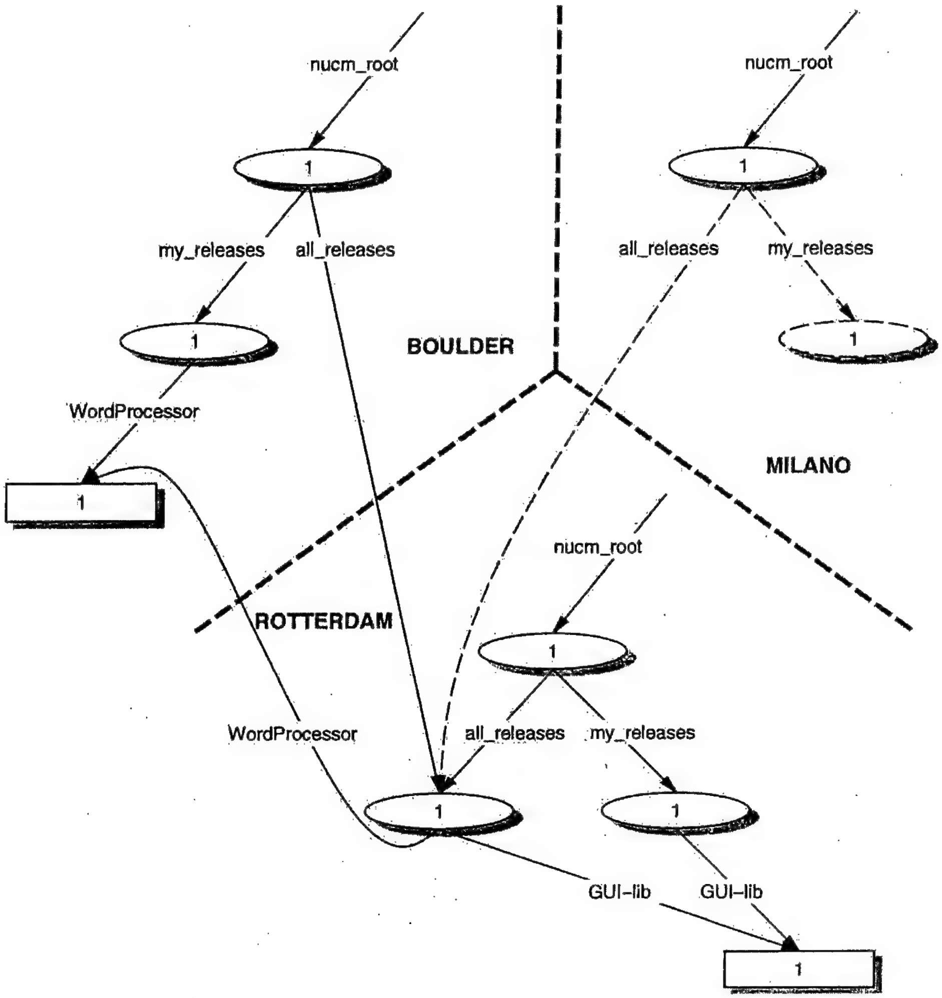


Fig. 11. Federated SRM repository after Milano joins the federation.

by using the function nc_initiatechange and, subsequently, uses the function nc_add to add the newly created collection my_releases and the existing collection all_releases to the collection nucm_root. The collection nucm_root is then stored in the repository using the function nc_commitchangeandreplace and the workspace is finally removed by using the function nc_close. The result of all these actions is shown in Fig. 11. Assuming that the Milano site is being joined with the SRM repository of Rotterdam and Boulder, the dashed lines indicate the new artifact and the membership relationships that are created by the procedure join. Once the artifact and the relationships are created, Milano is a full part of the SRM federation; when it adds new releases to the repository, they can be accessed from all sites.

The main advantage in using NUCM to develop SRM is that distribution could be isolated. Only two lines in the example source code of Fig. 11 explicitly deal with distribution: In line 6 and line 7, the remote repository that contains the collection all_releases and the local repository that is going to join the SRM repository are explicitly identified. After that, all policy programming is, in fact, transparent with respect to distribution. This particularly exhibits itself in other portions of the SRM policy code. Adding or removing releases to the SRM repository can simply be programmed as additions and removals to the collections my_releases and all_releases since NUCM tracks the physical location of these collections. Similarly, through the collection all_releases, any site can retrieve releases from the other sites without having to know where a release is physically located.

This strength of isolating distribution exhibits itself throughout SRM. Only a small part of the complete implementation, less than two percent, explicitly deals with distribution. The remainder of the implementation is concerned with the actual functionality of SRM itself and, in fact, relies on the distribution transparency provided by the internal storage layer of SRM.

4.3 WebDAV

WebDAV [22], [52] is an emerging standard that proposes to add authoring and versioning primitives to the HTTP protocol [20]. In particular, the standard proposes extensions in the following five areas:

- *Metadata.* To be able to describe Web resources, WebDAV proposes the creation of new HTTP methods that add metadata to Web resources, as well as methods to query and retrieve the metadata.
- *Collections.* To be able to structure Web resources into higher-level constructs, WebDAV proposes the

creation of new HTTP methods that allow Web resources to be grouped into collections, as well as methods that change the membership of collections.

- *Name space management.* To be able to efficiently move, copy, and delete Web resources, WebDAV proposes the creation of new HTTP methods that manipulate the Web name space.
- *Locking.* To avoid multiple entities updating a single Web resource in parallel and, consequently, losing changes, WebDAV proposes the creation of new HTTP methods that allow Web resources to be locked and unlocked for exclusive write access.
- *Version management.* To be able to keep a history of Web resources, WebDAV proposes the creation of new HTTP methods that allow Web resources to be versioned.

Although the objective of WebDAV (i.e., providing an infrastructure for distributed authoring and versioning) is slightly different from the objective of NUCM (i.e., providing a distributed repository to construct configuration management policies), the interface methods that have been proposed for both are strikingly similar. Only two major differences exist. First, NUCM includes a naming model that explicitly incorporates a mechanism to refer to versions of artifacts, whereas the naming scheme of WebDAV does not define such a mechanism. Second, WebDAV specifies a particular versioning policy, namely, the RCS-like lockable version tree, whereas NUCM is generic with respect to versioning policies.

Because of the similarity between NUCM and WebDAV, it seems advantageous to use NUCM to implement WebDAV, at least in order to quickly determine the utility of its policy. To this end, we created a simple HTTP server that is also a NUCM client. Most of the new HTTP methods translate into direct calls to the NUCM interface, but some require more work. In particular, the versioning routines of WebDAV prescribe a policy that is based on a version tree. To implement this tree, we have to map the versions in the tree to versions of NUCM artifacts. In our implementation, this mapping is created by storing two NUCM artifacts for each WebDAV artifact, namely, the actual artifact and an associated artifact that stores the version tree for that artifact. In addition, the version tree artifact has attributes associated with it that map each version number in the tree to a NUCM version number.

Fig. 12 shows how one of the procedures in our WebDAV implementation, namely, `checkin`, takes advantage of this approach. (Note that, because the WebDAV standard has continued to evolve, the example given here is not completely consistent with the current version of the standard.) The function stores a new version of an artifact and updates the version tree accordingly. Its functionality can be divided into five separate parts.

In the first part, several parameters used in the remainder of the function are determined. The names of the artifact being checked in and its corresponding version tree artifact are constructed first. Subsequently, the type of the artifact being checked in and its NUCM version number under which it was checked out are obtained.

In the second part of the function, the new version of the artifact is read from the WebDAV client and, subsequently, stored in the repository through the use of the function `nc_commitchange`.

The third part of the function serves an important role: It is the part that updates the version tree. We do not show the actual algorithm that determines the new version number since it does not involve any use of NUCM functions. Instead, it is shown how the version tree is obtained from the repository, updated with the new version information and stored back into the repository. Note the use of the function `nc_commitchangeandreplace` to replace the version tree since there is no need to store multiple versions of the version tree itself.

The fourth part of the function sets new attributes for some of the artifacts in the repository. In particular, it preserves the type of the artifact that was checked in and relates the version in the version tree with the NUCM version of the new artifact. Note that the type information is attached to the new version of the artifact, for which a new NUCM name is first constructed.

Finally, in the fifth part of the function, the artifact that was previously checked out and locked for modifications is unlocked so that other users can now modify this version.

Once again, the reusability of NUCM proved to be valuable in the implementation of WebDAV. The total number of lines of source code that were developed to create a prototype WebDAV implementation was only 1,500, of which approximately 40 percent accounts for a graphical user interface that can be used to perform WebDAV operations.

Admittedly, our experimental implementation does not cover all the functionality of WebDAV. However, the limited amount of code that needed to be developed and the minimal time required to produce that code together demonstrate an important aspect of NUCM: It can be used to quickly demonstrate prototype CM policies. The development of a standard like WebDAV can particularly benefit from such an approach since the ramifications of specific policy decisions can be explored with an early implementation.

5 Related Work

In its many years of existence, the discipline of configuration management has produced numerous industrial and research systems. Some provide only version control facilities, (e.g., RCS [47], SCCS [41], Sablime [5]), others provide more complete configuration management solutions (e.g., CVS [7], CoED [6], Perforce [38]), and yet others provide integrated environments that incorporate process management and/or problem tracking facilities (e.g., Adele [17], ClearCase [3], Continuus [12]). With respect to distribution, some of the CM systems are only suited for use at a single site (e.g., EPOS [35], ShapeTools [31], SourceSafe [32]), others incorporate a simple, sometimes Web-based, client-server interface (e.g., DCVS [7], Perforce [38], WWCM [25]), and yet others provide more advanced distribution mechanisms such as replication (e.g., ClearCase Multisite [2], Continuus DCM [13], PVCS SiteSync [28]). To understand the position of the abstraction layer and its implementation in NUCM in this large space of CM systems, we examine the evolution of CM system architectures.

Fig. 13a shows the architecture that has traditionally been used: A CM system is constructed as a single, monolithic entity that tightly integrates its storage mechanism with its CM policy. This approach is still the way in which most

```
  int checkin(const char* path, const char* oldversion, FILE* client)
  {
     //
     // Part 1: Determine necessary information.
     //
     sprintf(tree, "//%s:%s/nucm_root/TREE/%s", NUCMHOST, NUCMPORT, path);
     sprintf(treefilename, "%s/%s", WORKSPACE, (rindex(tree, '/')+1));
     sprintf(artifact, "//%s:%s/nucm_root/%s:%s", NUCMHOST, NUCMPORT, path, nucmversion);
     sprintf(filename, "%s/%s", WORKSPACE, (rindex(artifact, '/')+1));
*    nc_getattributevalue(tree, "", oldversion, nucmversion);
*    nc_getattributevalue(artifact, "", "TYPE", type);

     //
     // Part 2: Store new version of the artifact.
     //
*    nc_open(artifact, "", WORKSPACE, "");
*    nc_initiatechange(filename, "");
     fd = open(filename, O_TRUNC — O_WRONLY);
     while ((n = fread(bytes, 4096, 1, client)) ¿ 0)
        write(fd, bytes, n);
     close(fd);
*    nc_commitchange(filename, "", newnucmversion);
*    nc_close(filename, "", 0);

     //
     // Part 3: Update the version tree.
     //
*    nc_open(tree, "", WORKSPACE, "");
*    nc_initiatechange(treefilename, WORKSPACE);
     fd = fopen(treefilename, "r+");
     ...
     ... /* Determine new version number */
     ...
     sprintf(line, "%s CHILD OF %s“n”, newversion, oldversion);
     fputs(line, fd);
     fclose(fd);
*    nc_commitchangeandreplace(filename, "");
*    nc_close(filename, "", 0);

     //
     // Part 4: Set new attributes.
     //
     sprintf(newartifact, "//%s:%s/nucm_root/%s:%s", NUCMHOST, NUCMPORT, path, newnucmversion);
*    nc_testandsetattribute(tree, "", newversion, newnucmversion);
*    nc_testandsetattribute(newartifact, "", "TYPE", type);

     //
     // Part 5: We are done, unlock the artifact.
     //
*    nc_removeattribute(artifact, "", "LOCK");
  }
```

Fig. 12. WebDAV routine to check in a file.

CM systems are built, as exemplified by CoED [6] and DSCS [33], both of which were only recently developed. The advantage of this architecture is that it allows a CM system to optimize its storage to precisely match the needs of the CM policy. A clear disadvantage, however, is that a resulting CM system tends to be rather inflexible [15]. Moreover, such a CM system typically has to be constructed from the ground up, which is a major effort even today.

As CM systems have become more advanced, some have turned towards using a commercial, generic database management system as the underlying storage mechanism (e.g., ClearCase [3], Continuus [12], TrueCHANGE [45]). Illustrated in Fig. 13b, the advantage of this solution is that the database management system provides a reliable and reusable platform that offers such services as transactions, concurrency control, and rollbacks. These services no longer have to be implemented by the organization that develops the CM system.

Following the pattern of providing an increased level of abstraction with which to build CM systems, Fig. 13c illustrates that NUCM represents the next step in this evolutionary pattern. Although NUCM currently does not provide the same level of robustness and reliability as a database, the abstraction layer that it provides has the advantage of being highly specialized towards configuration management. Thus, as compared to a generic database, the model and interface defined by the abstraction layer raise the level of abstraction with which CM policies can be constructed and thereby facilitate their rapid implementation. As stated, NUCM currently lacks such essential

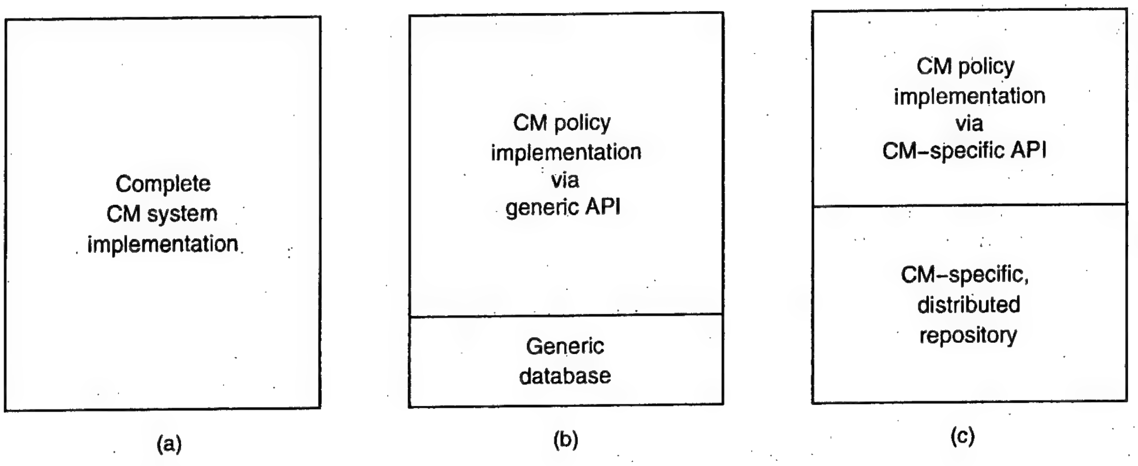


Fig. 13. Evolution of CM system architectures.

services as transactions, rollbacks, and caching, but it is anticipated that these can be incorporated in future implementations. Then, actual CM systems can be implemented based on the abstraction layer defined in this paper, while exhibiting the same qualities as a CM system built on top of a traditional database.

NUCM is only one of several systems that fall into the category of Fig. 13c. The other systems are CME [25], Gradient [4], CoMa [51], and ScmEngine [10].

CME extends RCE [26], itself a programmatic interface to RCS [47], with collection management. CME is similar to NUCM in that it provides an architectural separation of the repository from the actual system that stores and versions the artifacts. However, two significant differences exist. First, the programmatic interface of CME is not generic with respect to CM policies since it only contains functions that implement the check-in/check-out policy. The second difference is that CME is not distributed since it only interfaces to a single repository at a single site. Thus, whereas NUCM provides support for the construction of a variety of distributed CM policies, CME only provides support for the construction of centralized CM policies that are based on the check-in/check-out model.

Gradient is a CM repository that is based on automatic replication. Each update that is made to an artifact is broadcast instantly as a delta to all replicas. Because Gradient only allows incremental modifications to the artifacts it manages and, furthermore, assumes that modifications are independent of each other, it permits simultaneous updates to a single artifact at multiple sites. Gradient is similar in spirit to NUCM in that it provides an architectural separation of the storage mechanism from the CM system that uses it. But, as with CME, Gradient only supports a specific policy, both with respect to distribution (where it only supports replicated repositories), as well as with respect to CM policy (where it only supports the check-in/check-out policy).

CoMa is perhaps the one system that is closest in nature to the functionality provided by NUCM. CoMa introduces graph rewriting as a method of constructing specific CM policies. Based on a composition model, it utilizes graph rewriting rules to assert and enforce constraints. These constraints govern the evolution of the artifacts that are managed. The goal of CoMa is to evolve the interrelated sets of heterogeneous artifacts that are created throughout the software life cycle. Naturally, it therefore shares some of its goals with NUCM. Specifically, it needs to manage different kinds of artifacts and it needs to tailor its CM policy to the artifacts that are managed. As compared to NUCM, however, CoMa is limited in that it only supports the construction of variations of the check-in/check-out policy. Moreover, it does not support the distribution of artifacts over multiple physical locations. Thus, even though CoMa is more generically applicable than CME, it is similarly limited in that it only supports a small number of centralized CM policies.

ScmEngine is a distributed CM repository based on the X.500 directory protocol [39]. X.500 directory entries contain metadata describing the artifacts that are stored in physical repositories. Access servers leverage the standard X.500 directory protocol to create a logical repository that can be accessed by CM client programs. This distribution mechanism is, in essence, the same as the one defined by the distribution model of the abstraction layer. However, the remainder of the repository model and the programmatic interface provided by ScmEngine are significantly weaker than the ones defined by our abstraction layer. The repository model does not include collections and lacks the concept of version qualifiers to navigate in the version space. Moreover, the programmatic interface is very specific and lacks support for the construction of a wide variety of CM policies, only supporting the traditional check-in/check-out policy.

Outside the domain of configuration management, we can identify groupware and versioned databases as two important lines of work that are closely related to the work presented in this paper. In groupware, the need for distribution, versioning, and workspaces seems to imply that our abstraction layer could be appropriate for use in constructing a groupware system. However, this is not so. Whereas the abstraction layer is based on the principle of workspaces that provide isolation from changes by others, groupware systems tend to focus on collaborative workspaces [16], [21]. The set of issues involved in supporting each is rather different and, consequently, we believe groupware, even though related, falls outside of the domain of NUCM.

Versioned databases (e.g., ODE [1], TVOO [42]) are related to NUCM since NUCM itself can be viewed as a versioned database. In fact, many of the features of NUCM are shared by versioned databases. However, an important difference exists, which is the presence of a specific repository model and its associated programmatic interface. Whereas these are generic in nature in a versioned database (e.g., an entity relationship model with SQL), both are highly specialized by our abstraction layer. In essence, one could consider the abstraction layer that is incorporated in NUCM to be a layer on top of a versioned database that implements a particular schema (the repository model) and provides a number of standard views and operations (the programmatic interface).

6 Conclusion

For the past few years, the field of configuration management has been in a consolidation phase with the research results of the 1980s being transferred to the commercial products of the 1990s. Nonetheless, new CM systems are still being proposed and constructed. Some of these are new entries in an increasingly competitive marketplace. Others implement proprietary solutions that are tailored to the situation at hand. Yet others explore new ground and form the basic research that will lead to the next generation of CM systems. During their design and implementation, though, all face what we consider to be one of the most pertinent problems in the field of configuration management: No suitable platform exists that can serve as a flexible testbed for the rapid construction of potentially distributed CM systems.

Based on the critical observation that, to effectively address this problem it is necessary to separate CM repositories from CM policies, this paper has introduced a novel abstraction layer that represents a first step towards addressing this problem. The abstraction layer precisely defines a generic model of a distributed repository and a programmatic interface for implementing, on top of the repository, specific CM policies. Characteristics of the abstraction layer are its policy independence, its ability to manage a wide variety of different kinds of artifacts, its inherent distributed operation, and its ability to support traditional CM functionality.

The abstraction layer was designed to facilitate the rapid construction of, and experimentation with, CM policies. However, it has proven to facilitate more than that. DVS and SRM, two of the systems that were originally constructed to demonstrate the applicability and flexibility of the abstraction layer, have evolved into complete CM systems. Both are now in use in settings that involve multiple parties in multiple geographical locations and both continue to evolve with respect to the functionality they provide. The abstraction layer, thus, not only supports the construction of new CM policies, but also their gradual evolution into more mature systems.

Our work does not end here. Although we certainly believe that the abstraction layer is a step in the right direction towards providing a generic, reusable, and distributed platform for CM policy programming, much work remains to be done. In particular, it is our belief that the abstraction layer facilitates the construction of standard policy libraries, thereby even further reducing the effort of implementing a CM system. Moreover, we expect to be able to use the abstraction layer as a vehicle for exploring other important problems in configuration management. For example, we believe the abstraction layer provides a suitable platform for investigating the problems of CM policy integration [37].

Acknowledgments

This work was supported in part by the Air Force Materiel Command, Rome Laboratory, and the Advanced Research Projects Agency under Contract Numbers F30602-94-C-0253 and F30602-98-2-0163. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

References

[1] R. Agrawal, S. Buroff, N.H. Gehani, and D. Shasha, "Object Versioning in ODE," *Proc. Seventh Int'l Conf. Data Eng.*, pp. 446–455, Apr. 1991.

[2] L. Allen, G. Fernandez, K. Kane, D. Leblang, D. Minard, and J. Posner, "ClearCase MultiSite: Supporting Geographically-Distributed Software Development," *Software Configuration Management: Int'l Conf. Software Eng. SCM-4 and SCM-5 Workshops Selected Papers*, pp. 194-214, 1995.

[3] Atria Software, *ClearCase Concepts Manual*. Natick, Mass., 1992.

[4] D. Belanger, D. Korn, and H. Rao, "Infrastructure for Wide-Area Software Development," *Proc. Sixth Int'l Workshop Software Configuration Management*, pp. 154-165 1996.

[5] Bell Labs, Lucent Technologies, *Sablime v5.0 User's Reference Manual*. Murray Hill, New Jersey, 1997.

[6] L. Bendix, P.N. Larsen, A.I. Nielsen, J.L.S. Petersen, "CoED—A Tool for Versioning of Hierarchical Documents," *Proc. Eighth Int'l Symp. System Configuration Management*, 1998.

[7] B. Berliner, "CVS II: Parallelizing Software Development," *Proc. 1990 Winter USENIX Conference*, pp. 174-187, 1990.

[8] C. Burrows and I. Wesley, *Ovum Evaluates Configuration Management*. Burlington, Mass.: Ovum Ltd., 1998.

[9] A. Carzaniga, *DVS 1.2 Manual*. Dept. of Computer Science, Univ. of Colorado, Boulder, June 1998.

[10] J.X. Ci, M. Poonawala, and W.-T. Tsai, "ScmEngine: A Distributed Software Configuration Management Environment on X.500," *Proc. Seventh Int'l Workshop Software Configuration Management*, pp. 108-127, 1997.

[11] P.C. Clements and N. Weiderman, "Report on the Second International Workshop on Development and Evolution of Software Architectures for Product Families," Technical Report SEI-98-SR-003, Software Eng. Inst., Pittsburgh, Penn., May 1998.

[12] Continuus Software Corporation, *Continuus Task Reference*. Irvine, Calif., 1994.

[13] Continuus Software Corporation, *Distributed Code Management for Team Engineering*. Irvine, Calif., 1998.

[14] S. Dart, "Concepts in Configuration Management Systems," *Proc. Third Int'l Workshop Software Configuration Management*, pp. 1–18, 1991.

[15] S. Dart, "Not All Tools are Created Equal," *Application Development Trends*, vol. 3, no. 10, pp. 39–54, Oct. 1996.

[16] A. Dix, T. Rodden, and I. Sommerville, "Modeling the Sharing of Versions," *Proc. Sixth Int'l Workshop Software Configuration Management*, pp. 282-290, 1996.

[17] J. Estublier and R. Casallas, "The Adele Configuration Manager," *Configuration Management, Trends in Software*, W. Tichy, ed., no. 2, pp. 99–134, 1994.

[18] P.H. Feiler, "Configuration Management Models in Commercial Environments," Technical Report SEI-91-TR-07, Software Eng. Inst., Pittsburgh, Penn., Apr. 1991.

[19] P.H. Feiler and G. Downey, "Transaction-Oriented Configuration Management: A Case Study," Technical Report CMU/SEI-90-TR-23, Software Eng. Inst., Pittsburgh, Penn., 1990.

[20] R. Fielding, J. Gettys, J.C. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol–HTTP/1.1," Internet Proposed Standard RFC 2068, Jan. 1998.

[21] P. Fröhlich and W. Nejdl, "WebRC: Configuration Management for a Cooperation Tool," *Proc. Seventh Int'l Workshop Software Configuration Management*, pp. 175-185, 1997.

[22] Y.Y. Goland, E.J. Whitehead, Jr., A. Faizi, S. Carter, and D. Jensen, "HTTP Extensions for Distributed Authoring—WEBDAV," Internet Proposed Standard RFC 2518, Feb. 1999.

[23] G.T. Heineman, "A Transaction Manager Component for Cooperative Transaction Models," PhD thesis, Columbia Univ., Dept. of Computer Science, New York, June 1996.

[24] T. Hung and P.F. Kunz, "UNIX Code Management and Distribution," Technical Report SLAC-PUB-5923, Stanford Linear Accelerator Center, Stanford, Calif., Sept. 1992.

[25] J.J. Hunt, F. Lamers, J. Reuter, and W.F. Tichy, "Distributed Configuration Management via Java and the World Wide Web," *Proc. Seventh Int'l Workshop Software Configuration Management*, pp. 161-174, 1997.

[26] J.J. Hunt and W.F. Tichy, *RCE API Introduction and Reference Manual.* Germany: Xcc Software, 1997.

[27] J.J. Hunt, K.-P. Vo, and W.F. Tichy, "Delta Algorithms: An Empirical Analysis," *ACM Trans. Software Eng. and Methodology*, vol. 7, no. 2, pp. 192–214, Apr. 1998.

[28] INTERSOLV, *PVCS VM SiteSync and Geographically Distributed Development.* Rockville, Md., 1998.

[29] R. Leung, "Versioning on Legal Applications Using Hypertext," *Proc. Workshop Versioning in Hypertext Systems*, Sept. 1994.

[30] Y.-J. Lin and S.P. Reiss, "Configuration Management with Logical Structures," *Proc. 18th Int'l Conf. Software Eng.*, pp. 298–307, Mar. 1996.

[31] A. Mahler and A. Lampen, "An Integrated Toolset for Engineering Software Configurations," *Proc. ACM SOFSOFT/SIGPLAN Software Eng. Symp. Practical Software Eng. Environments*, pp. 191–200, Nov. 1988.

[32] Microsoft Corporation, *Managing Projects with Visual SourceSafe*, Redmond, Wash., 1997.

[33] B. Milewski, "Distributed Source Control System," *Proc. Seventh Int'l Workshop Software Configuration Management*, pp. 98-107, 1997.

[34] Mortice Kern Systems, Inc., *Untangling the Web: Eliminating Chaos*, Waterloo, Canada, 1996.

[35] B.P. Munch, "Versioning in a Software Engineering Database—the Change-Oriented Way," PhD thesis, DCST, NTH, Trondheim, Norway, Aug. 1993.

[36] B. O'Donovan and J.B. Grimson, "A Distributed Version Control System for Wide Area Networks," *Software Eng. J.*, Sept. 1990.

[37] F. Parisi-Presicce and A.L. Wolf, "Foundations for Software Configuration Management Policies Using Graph Transformations," *Proc. 2000 Conf. Foundational Aspects of Software Eng.*, Mar. 2000.

[38] Perforce Software, *Networked Software Development: SCM over the Internet and Intranets.* Alameda, Calif., Mar. 1998.

[39] Recommendation X.500 (08/97)–Information Technology–Open Systems Interconnection–the Directory: Overview of Concepts, Models and Services, Aug. 1997.

[40] R.J. Ray, "Experiences with a Script-Based Software Configuration Management System," *Software Configuration Management: Int'l Conf. Software Eng. SCM-4 and SCM-5 Workshops Selected Papers*, 1995.

[41] M.J. Rochkind, "The Source Code Control System," *IEEE Trans. Software Eng.*, vol. 1, no. 4, pp. 364–370, Dec. 1975.

[42] L. Rodriguez, H. Ogata, and Y. Yano, "An Access Mechanism for a Temporal Versioned Object-Oriented Database," *IEICE Trans. Information and Systems*, vol. E82–D, no. 1, pp. 128–135, Jan. 1999.

[43] R.A. Smith, "Analysis and Design for a Next Generation Software Release Management System," Master's thesis, Univ. of Colorado, Boulder, Dec. 1999.

[44] Softool Corp., *CCC/Manager, Managing the Software Life Cycle across the Complete Enterprise*, Goleta, Calif., 1994.

[45] Software Maintenance & Development Systems, Inc., *Aide de Camp Product Overview*, Concord, Mass., Sept. 1994.

[46] SQL Software, *The Inside Story: Process Configuration Management with PCMS Dimensions*, Vienna, Va., 1998.

[47] W.F. Tichy, "RCS, A System for Version Control," *Software—Practice and Experience*, vol. 15, no. 7, pp. 637–654, July 1985.

[48] A. van der Hoek, "A Generic, Reusable Repository for Configuration Management Policy Programming," PhD thesis, Dept. of Computer Science, Univ. of Colorado, Boulder, Jan. 2000.

[49] A. van der Hoek,, R.S. Hall, D.M. Heimbigner, and A.L. Wolf, "Software Release Management," *Proc. Sixth European Software Eng. Conf.*, pp. 159-175, Sept. 1997.

[50] A. van der Hoek,, D.M. Heimbigner, and A.L. Wolf, A Generic, Peer-to-Peer Repository for Distributed Configuration Management," *Proc. 18th Int'l Conf. Software Eng.*, pp. 308–317, Mar. 1996.

[51] B. Westfechtel, "A Graph-Based System for Managing Configurations of Engineering Design Documents," *Int'l J. Software Eng. and Knowledge Eng.*, vol. 6, no. 4, pp. 549–583, 1996.

[52] E.J. Whitehead, Jr., "World Wide Web Distributed Authoring and Versioning (WebDAV): An Introduction," *StandardView*, vol. 5, no. 1, pp. 3–8, Mar. 1997.

[53] A. Zeller and G. Snelting, "Unified Versioning Through Feature Logic," *ACM Trans. Software Eng. and Methodology*, vol. 6, no. 4, pp. 398–441, Oct. 1997.

André van der Hoek received a joint BS and MS degree in business-oriented computer science from the Erasmus University, Rotterdam, and the PhD degree in computer science from the University of Colorado at Boulder. He is an assistant professor in the Department of Information and Computer Science and a faculty member of the Institute for Software Research, both at the University of California, Irvine. His research interests include configuration management, software architecture, configurable distributed systems, and software education. He has developed several CM systems, was a cochair of the Ninth International Symposium on System Configuration Management, and is chair of the Tenth International Workshop on Software Configuration Management. He is a member of the IEEE and the IEEE Computer Society.

Antonio Carzaniga received the Laurea degree in electronic engineering and the PhD degree in computer science from Politecnico di Milano, Italy. He is currently a research associate with the Department of Computer Science at the University of Colorado at Boulder. His research interests are in the areas of distributed systems engineering, software engineering, computer networks, content-based routing, middleware, programming languages, and software engineering tools.

Dennis Heimbigner received the PhD degree in computer science from the University of Southern California, Los Angeles. He is a research faculty member in the Department of Computer Science, University of Colorado at Boulder. Prior to that, he was at TRW in Redondo Beach, California. His current research interests include distributed computing, peer-to-peer computing, and configuration management. He has published papers in the areas of configuration management, distributed computing, software process, software development environments, concurrent programming, and programming language semantics. He is a member of the IEEE and the IEEE Computer Society.

Alexander L. Wolf received the PhD degree in computer science from the University of Massachusetts at Amherst. He is a faculty member in the Department of Computer Science, University of Colorado at Boulder. Previously, he was at AT&T Bell Laboratories in Murray Hill, New Jersey. His research interests are in the discovery of principles and development of technologies to support the engineering of large, complex software systems. He has published papers in the areas of software engineering environments and tools, software process, software architecture, configuration management, distributed systems, and persistent object systems. Dr. Wolf served as program cochair of the 2000 International Conference on Software Engineering (ICSE 2000), is currently serving as vice chair of the ACM Special Interest Group in Software Engineering (SIGSOFT), and is on the editorial board of *ACM Transactions on Software Engineering and Methodology* (TOSEM). He is a member of the IEEE Computer Society.

A Tamper-Detecting Implementation of Lisp

Dennis Heimbigner
Computer Science Department
University of Colorado, Boulder, CO 80309-0430, USA
Dennis.Heimbigner@colorado.edu
303-492-6643 (voice) 303-492-2844 (fax)

Abstract

An important and recurring security scenario involves the need to carry out trusted computations in the context of untrusted environments. It is shown how a tamper-detecting interpreter for a programming language – specifically Lisp 1.5 – combined with the use of a secure co-processor can address this problem. The term "tamper-detecting" means that any attempt to corrupt a computation carried out by a program in the language will be detected on-line and the computation aborted. This approach executes the interpreter on the secure co-processor while the code and data of the program reside in the larger memory of an associated untrusted host. This allows the co-processor to utilize the host's memory without fear of tampering even by a hostile host. This approach has several advantages including ease of use and the ability to provide tamper-detection for any program that can be constructed using the language.

Keywords: co-processor, anti-tamper, Lisp, on-line

1. Computing in a Hostile Environment

An important and recurring security scenario involves the need to carry out trusted computations in the context of untrusted environments. One approach is to combine a secure co-processor [6][15] with an untrusted host computer. The secure co-processor provides the environment in which to perform trusted computations, and the insecure host provides additional resources that may be used by the trusted processor. Unfortunately, there is no guarantee that the host will not tamper with the resources used by the secure co-processor in an attempt to corrupt the operation of the secure co-processor.

This paper demonstrates a solution where a programming language system – specifically Lisp 1.5 – is used to provide a convenient and general mechanism for tamper-detecting utilization of a specific resource, namely the memory of an untrusted host. An interpreter for the language system resides on the secure co-processor, but the programs and data executed by the interpreter reside in the memory of the untrusted host.

In this context, the term "tamper-detecting" means that any attempt to corrupt a computation carried out by a program in the language will be detected on-line (before the computation is complete), and the computation will be aborted.

In order to limit the scope of the problem, only the issue of integrity is addressed in this paper; the issue of confidentiality is deferred. It seems reasonable, however, to assume that adding confidentiality is a straightforward application of encryption to the values stored in the host memory.

2. Why Lisp 1.5?

Lisp 1.5 was chosen as the target programming language primarily because of its simplicity and to demonstrate proof-of-concept. Lisp provides a simple, usable, and complete language. It has a small interpreter [11] that can easily be implemented on a secure co-processor with limited resources. Equally important, Lisp uses lists as its only data structure, both for programs and for data; hence tamper detection can be applied to both code and data with no extra effort. Thus Lisp provides a good platform for exploring issues in tamper-detecting language implementations.

3. Lisp List Representation

This paper assumes familiarity with the Lisp 1.5 language and its implementation. A complete review of the language and its original implementation is available in the "Lisp 1.5 Programmer's Manual" [8] by John McCarthy et al.

Briefly reviewing, Lisp lists are composed of cells linked via pointers. A standard Lisp cell consists of three fields: (1) Car and (2) Cdr pointers to other cells and (3) a flag field indicating properties of the cell. Traditionally, the "list" is considered to be the set of cells reached by following the Cdr pointers. Cells reached through the Car pointer are often referred to as "sublists". Cyclic lists are allowed, as are lists with common sublists. The standard flags are as follows.

1. ATOM – the cell format is that of an atomic (i.e., non-list cell) value.
2. NUMBER – a subclass of ATOM indicating that this cell holds a numeric value.
3. FREE – this cell is on the *freelist*.
4. MARK, CARCHAIN, and CDRCHAIN – for use during garbage collection (see Section 7).

In the implementation used here, the Car of an atom cell points to the name represented as a list of integers, where each integer is the value of a character in the name. The Cdr field of an atom is used to link it into a list of all known atoms (the OBLIST). We assume that atoms can be created but never destroyed. The root of the OBLIST is one of several special pointers kept in the memory of the secure co-processor where they are immune from insecure modification. The value of a non-integer atom is handled using the traditional ALIST and APVAL mechanisms. Nil is a special atom whose value is itself and which is traditionally used to terminate lists. Numeric atoms contain the integer value in the Cdr field. This makes the reasonable assumption that an integer and a pointer have the same size.

4. Attack and Trust Assumptions

The critical trust assumption is that any values kept in the memory of the secure co-processor cannot be directly read or modified by the untrusted host. Thus the code and data on the secure co-processor constitute the trusted computing base for the Lisp implementation. The only assumed attack mechanism by which the host can tamper with a computation of the secure co-processor is through the values the host returns in response to read requests from the secure co-processor. Specifically not addressed are any physical attacks against the secure co-processor.

5. Secure Co-Processor – Host Access Protocol

The secure co-processor accesses the host through the following primitive operations.

- *read(i)* – return the content of host memory location *i*.
- *write(i,c)* – write *c* as the new content for host memory location *i*.
- *alloc(n)* – allocate *n* sequentially located cells of new host memory and return the address of the start of that memory.
- *release()* – let the host reclaim all allocated memory.

To simplify the Lisp interpreter in the co-processor, the above operations are wrapped by the following Lisp-oriented operations.

- *CAR(p), CDR(p)* – read the cell (with tamper detection) pointed to by *p* and extract either the Car or Cdr field respectively.
- *CONS(p,q,f)* – construct the content of a cell with *p* in its Car field, with *q* in its Cdr field, and with *f* in its flag field. Then obtain a pointer to a free cell from the *freelist* and write (with tamper detection) the newly constructed cell into that free cell.
- *new()* – return a pointer to an unused cell. It is assumed that the cell comes from the *freelist*, which is a special list of unused cells linked together through their Cdr fields and with the special FREE flag set. Cells are checked for tampering when they are removed from the freelist. If all space is exhausted, then an *alloc()* request is made to the host to obtain more memory to construct a new freelist.

6. Basic Elements of Tamper-Detection

Tamper-detection is achieved by dividing the whole computation (the program execution) into *epochs*. Each epoch has an associated index that acts as a timestamp for all write operations performed during that epoch. Epoch boundaries are defined by occurrences of garbage collection. Thus, every time the garbage collector is invoked, a new epoch begins. The sequence of epochs continues until the computation is complete.

Tampering with a cell's content is detected by adding a cryptographic signature as a new field in each cell (the σ field in Figure 1). The signature is computed using any reasonable collision/computation resistant (i.e., one-way and hard to invert) and second pre-image resistant hash function such as SHA-1 [9] or MD5 [9]. The hash function takes the ordered concatenation of the following four values as its input.

- Cell content – the Car, Cdr, and Flags fields (also ordered).
- Cell address – the address from which the cell was read.
- Time stamp – the current epoch index.
- Secret key – a key known only to the secure co-processor.

Whenever a cell is read, the signature is recomputed and if it matches the stored signature, then it is assumed that the Car, Cdr, and Flag fields are valid. The σ signature field allows the content of a cell to be validated using only the address of the cell, the content of the cell, and the secret key and epoch index information contained in the secure co-processor. Note that the signature, by itself, does not prevent replay attacks, only synthesis attacks.

The epoch indices need not be sequential since only two are ever used at any point in time, and then only during garbage collection. So, the epoch index can be any non-repeating sequence of random numbers. This suggests that the need for the epoch index can be replaced by using the secret key instead. This has the advantage of introducing a new secret key for every epoch, which provides a natural mechanism for re-keying. In subsequent discussion, it is assumed that the epoch index and the secret key are combined into a single *epoch key*.

It is important to note that in the absence of encryption, the security parameter for this signature is not determined by the total input size (512 bits), but rather by the size of the secret key (128 bits). As a consequence, brute force attacks on the signature are possible, but are assumed to be hard. This is in line with prior work [4], which assumes the adversary has limited computational power. Information theoretic bounds [1][3] are not considered here.

While the signature prevents synthesis, replay is prevented by enforcing the *write-once-per-epoch* property. This means that during an epoch, any given cell in the host memory will be written at most once. This property is enforced by the fact that the only memory writing that can occur is through the CONS procedure, and that operator is defined to always store its result in a new cell taken from the freelist.

Within an epoch, a cell will have at most two values as its content. For cells that are already allocated at the beginning of the epoch, their content will never change. For cells that are on the freelist, their initial content is the initial value as a member of the freelist. The second is the value written into it at the time it is allocated. Thus for any cell, the only possible replay attacks are the following:

1. Replay the cell content from another epoch,
2. Replay the content of some other cell in the same epoch,
3. Replay the content of an allocated cell as it was when it was on the freelist.

Since the signature includes the cell address and the epoch key, cases 1 and 2 can be detected by failure to validate the signature when the cell is read from the host processor. Case 3 will be detected by the presence of a FREE flag, which cannot occur when reading a cell reachable by any pointer (other than the head of the freelist). Thus the only cell value that an attacker can return is the correct value of the cell as written (once) during the epoch.

The write-once property has some important consequences. In particular, it disallows use of traditional extensions to Lisp such as REPLACA, REPLACD, and PROG because they support direct cell modifications. Write-once does not prevent lambda binding (using an ALIST) and SETQ (using an APVAL list); see the longer technical report [6] for details.

7. Epoch Transition by Garbage Collection

The transition from one epoch to the next is tied to garbage collection. Garbage collection is expected to have two specific effects upon its completion.

- All unreachable cells have been linked into a single freelist.
- All reachable cell signatures have been updated based on the new epoch key.

The garbage collection phase violates the write-once-per-epoch assumption and so it offers significant opportunities for tampering. Replay attacks are especially tempting because each cell will be written several times.

In the following discussion, familiarity is assumed with the common approaches to garbage collection. In particular, familiarity is assumed with the standard mark-and-sweep approach, which was chosen because it isolates the collection activity into a single phase for which special anti-tampering mechanisms can be used. Knowledge of the well-known Schorr-Waite(-Deutsch) [12] algorithm for marking is also assumed. This algorithm was chosen because it avoids the need for a separate stack. As it performs its depth-first walk, this algorithm temporarily reverses the list structure of the lists on the current path of the walk.

The mark phase operates by doing a depth-first traversal of the graph of cells reachable from a defined set of root pointers kept in the secure co-processor. As each cell is first reached, it is marked. At the point where a cell is last touched in the walk, its content is re-signed using the new epoch key. Thus at the end of the traversal all reachable cells have been touched and re-written. Since they have been re-signed using the new epoch key, they have effectively been moved into the new epoch. A given path ends when it encounters an atom or encounters a cell that has already been marked. This latter case can occur either because some cells may be reachable by more than one path during the walk or because the list is cyclic. Cells C4 and C5 in Figure 1 show these two cases respectively.

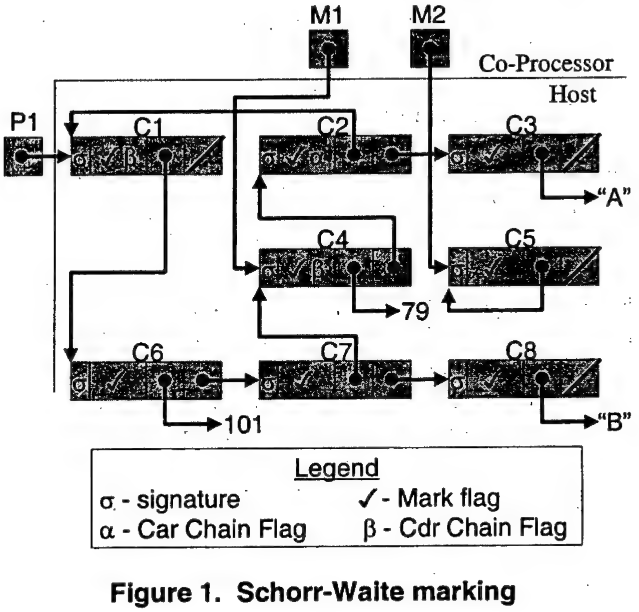


Figure 1. Schorr-Waite marking

During the marking process, a cell can be in one of four states.

(1) Unmarked – any cell not yet reached during marking will be in the just completed epoch and no garbage collector related flags will have been set in the cell.

(2,3) Car or Cdr Chaining – some cells on the current depth-first path will have their Car or Cdr fields reversed and will have a flag set to indicate that fact. In addition, such a cell will have its MARK flag set. It is still considered to be in the just completed epoch.

(4) Complete – any cell for which Car and Cdr chaining is completed will be in the new epoch and will have its MARK flag set.

Figure 1 shows a point in the traversal of a set of lists. The dotted lines indicate the boundary between the secure co-processor and the host. The box labeled P1 at the left represents a root pointer in the secure co-processor. The boxes labeled M1 and M2 represent special pointers in the secure co-processor. They are used to track the state of the marking procedure. Thus, M1 points to the last cell in the current depth-first path (C4) and M2 points to the next cell to be marked (C5). Note the reversal of several pointers in cells C1, C2, and C4 and the associated flags α (for Car reversal chaining) and β (for Cdr reversal chaining).

After marking is completed, the sweep phase examines every cell in the sequential order defined by its memory address. The chunks of *alloc()*'d memory are tracked using an special ALLOC list. If the cell is unmarked, then it is unused, and it is marked as a free cell and is linked to the freelist. In order to avoid a second sweep to reset the mark bits, the secure co-processor just inverts the sense of the mark bit so that in the next garbage collection, all cells will be considered unmarked.

8. Tamper Detecting Garbage Collection

The goals for garbage collection are three-fold:

1. Immediately detect attempts to modify a cell's content,
2. Detect replays no later than the end of garbage collection.
3. Detect replays before they can cause garbage collection to fail,

Thus, we are willing to allow replays to occur as long as they do not corrupt garbage collection, but in any case, replays must be detected before normal computation resumes.

The first goal is easily met if we continue to sign our cells every time we write them to the host memory. Again assuming that our hash function is hard to invert, we assume that attempts to modify a cell's content will fail. So at the start of garbage collection, a new secret epoch key is computed. During garbage collection, the σ signature field is recomputed every time a cell is modified. The specific epoch key, new or old, is chosen based on the step in the marking phase.

During Car and Cdr chaining, the cell is re-written using the old epoch key. When traversal of a cell is completed, the last action is to recompute its σ field to contain its signature but using the new epoch key.

When reading a cell, it is verified using the old epoch key if the cell appears unmarked, or appears to be involved in Car or Cdr chaining. Otherwise, it is verified using the new epoch key.

During the sweep phase, each cell is read in turn and, based on its content, is either ignored or linked to the freelist. Verification of the content of the cell depends on the flags field of the cell.

If the cell appears to be unmarked then its signature is validated using the old epoch key. If valid, then the cell is rewritten with a flag indicating that it is free. Its Cdr field is used to link it to the front of the freelist. The signature field of the free cell is computed using the revised content and using the new epoch key.

If the cell appears to be marked, then its signature is validated using the new epoch key, and otherwise it is left untouched.

The claim is that at the end of garbage collection, every cell is either on the freelist or has been marked. In both cases, the cell has been re-signed using the new epoch key. At this point, the next epoch starts and computation resumes.

9. Replay Attacks Against Garbage Collection

While a keyed, cryptographically strong hash function prevents the host from synthesizing corrupt cell content, it does not necessarily prevent replay attacks. Whenever a cell is read from the host memory during either the mark or sweep phases, a malicious host has the option of providing a replay of any of the four values stored in that cell during garbage collection.

1. It can replay the correct content of the cell.
2. It can replay the content of the cell as it was when involved in Cdr chaining.
3. It can replay the content of the cell as it was when involved in Car chaining.
4. It can replay the content of the cell as it was before garbage collection began.

Obviously Case 1 causes no problem. Cases 2 and 3 can only usefully occur during the mark phase because the sweep phase does not use the Car and Cdr chaining flags.

Even during the marking phase, these cases can only occur when it possible to reach a cell by more than one path. As figure 1 shows, this can occur for cells that are sublists of more than one list (cell C4) or are part of a cyclic list (cell C5). Under normal circumstances, this would cause the walk to encounter an already marked cell, which in turn would cause the walk to stop and back up to continue the walk down another path

If the host provides case 2 or 3 replay, this can cause no difficulties because the mark flag will be set and will cause the garbage collector to properly stop its marking and back up to a new depth-first path.

Case 4 is a problem both during the mark phase and the sweep phase. During marking, the legitimate content of the cell may indicate the cell has already been marked. If instead the host replays the old, unmarked cell content, then the garbage collector will happily re-mark that cell and everything reachable from it as long as the host replays unmarked cell values, possibly forever. This re-marking by itself causes no harm because marking is an idempotent operation.

The same thing may also happen during the sweep phase. That is, the host may replay the unmarked content of each cell. In this case, an active cell will be treated as a free cell and erroneously added to the freelist. Again, this causes no immediate harm since the cells of the freelist are never revisited during the sweep phase.

10. A Counting Solution

The only damaging effects of a Case 4 replay are to cause the mark phase to re-mark cells (possibly endlessly) or to cause the sweep phase to free cells that are really reachable. The effects of both attacks can be controlled using a counting technique.

Assume that at the beginning of garbage collection, we know Tc, the total number of allocated cells. In the untampered state:

$$Tc = Rc + Fc \qquad (1)$$

where Rc is the total number of reachable cells and Fc is the total number of free cells. In a possibly tampered state, we have the following inequality.

$$Tc \leq Mc + Sc \qquad (2)$$

where Mc is the number of unmarked cells seen during the mark phase and Sc is the number of unmarked cells seen during the sweep phase. The inequality is a consequence of the following two inequalities.

$$Rc \leq MC \quad (3) \qquad\qquad Fc \leq Sc \quad (4)$$

These two inequalities come from the following observations. First, if the attacker uses a case 4 replay during the mark phase, it will increase the number of apparently unmarked cells by 1, hence equation (3) will hold. If the attacker uses the same replay during the sweep phase, it will increase the number of apparently free cells by one, hence equation (4) will hold. Note that the number of marked cells can never be falsely increased because that would require the attacker to be able to mark a cell without detection, which is hypothesized to be impossible if our signature function is not invertible.

Using equations (1) and (2), it is then possible to detect the occurrence of tampering no later than the end of the sweep phase of garbage collection, which is our second goal. At that point, the number of marked cells plus the number of free cells will have been counted and if the total is greater than the total number of available cells (Tc), then tampering has occurred.

Our one remaining problem is the possibility of endless replay during the mark phase. If the attacker always replays unmarked cell values, then there is a potential for the mark phase to loop forever in the presence of any cyclic lists. To address this, we note one more equation.

$$Mc > Tc \Rightarrow Tc \leq Mc + Sc \tag{5}$$

This indicates that if the number of reachable cells Mc ever exceeds Tc, then equation (2) will of necessity be true and hence tampering must be occurring. Thus if we track the number of cells we mark, we are guaranteed to eventually detect a loop and signal a replay attack.

11. Preliminary Performance Measurements

An implementation of the tamper-detecting Lisp system has been completed. Preliminary performance measurements have been collected using g++ version2.95.3 on an Ultra-2 Sparc platform. These measurements must be interpreted carefully because of the platform and because of the following assumptions and implementation limitations underlying those measurements.

The most important limitation concerns signing overhead. We use a public domain, software-only implementation of the MD5 signing code from RSA with an average signing time of about 10 microseconds for signing 512 bits. This means that signing time dominates the current set of measurements and gives the appearance that the cost of tamper-detection is large. Given signing hardware and/or faster signing functions, it should be possible to reduce this cost to, say, 1 or 2 microseconds, at which point the signing cost becomes manageable.

The memory bandwidth between the secure co-processor and the host also impacts the overall performance, but it is not separately modeled in our performance measurements.. Not only is every cell read by the secure co-processor, the cell size increases because of the addition of the signature field. The communication channel between the host and the co-processor should be something like Firewire, USB 2.0, or a direct PCI bus connection to avoid bandwidth bottlenecks.

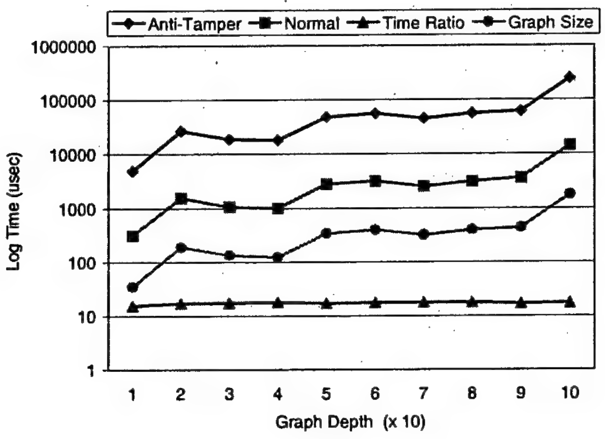


Chart 1. Graph marking times

With these limitations in mind, Chart 1 shows some preliminary performance measurements for doing the garbage collection mark phase on acyclic graphs whose depth ranges from 10 to 100. This particular measurement is included because tamper-detecting garbage collection, and marking in particular, is critical to the operation of the system. The list graphs to be marked are generated with random widths in each level (up to a maximum of 64 cells) and the number of roots is chosen randomly in the range 1 to 8. The Y-axis of the graph is the log of the elapsed time in microseconds and the X-axis is the graph depth.

The top line represents marking time when anti-tamper is activated. The second line from the top shows the normal marking time. The bottom line is the ratio (about 17), and the second line from the bottom line is the total number of cells in the graph.

In summary, the performance measurements appear to indicate that a feasible tamper-detection Lisp implementation can be constructed if certain bandwidth and signing speeds can be achieved. But detailed and accurate performance measurements must wait until the implementation is re-hosted onto real co-processor hardware.

12. Related Work

The approach proposed in this paper is directly inspired by the prior work in tamper-detecting data structures. These structures and implementing code are stored in the host's memory and have the property that any attempt by the host to tamper with the data structure will be detected. Examples of such data structures include random-access memory [3], simple linear lists [1][3], and stacks and queues [3][4].

The language approach has several advantages compared to the data structure approach. It is more general since the programmer can use any data structure that can be implemented in the language. Additionally, it hides the complexity of tamper-detection. Programmers do not have to worry about the problem of tampering because a solution is built into the language implementation and is inherited by all programs executed by the language interpreter. This solution also reduces and simplifies the code that must reside on the host processor. Data structure specific code is not required. Instead, the only required code is that necessary to allow the secure co-processor to read and write the host's memory and to request the allocation of blocks of the host's memory. Thus using the language approach, it should be easier to construct programs that can safely avail themselves of untrustworthy host memory.

This work explicitly assumes the use of secure hardware whose memory cannot be read or modified by the untrusted host. There exist software-only solutions [14] to the trusted computing problem that use various forms of obfuscation to prevent an untrusted software program from analyzing the actions of the trusted software. A recent theoretical result [2] casts doubt on the generality of this approach, and indicates that completely general software-only solutions may be impossible.

The use of *cryptopaging* [5][13] is another possible alternative to the approach presented here. Cryptopaging treats the host as secondary memory and the secure co-processor uses it as the target for paging its memory. Replay is prevented by maintaining a complete Merkle hash tree [10] whose leaves are the available pages of the host memory. The nodes of the Merkle tree are also kept in host memory. Cryptopaging has the advantage that it is fast because it uses modified hardware for signing and memory retrieval. This is also a disadvantage since our approach can use off-the-shelf hardware. Its other disadvantage is page-size. Our approach accesses memory in smaller chunks and uses language semantics to ensure that no replay occurs. The cryptopaging approach uses additional memory for storing the Merkle hash tree interior nodes in the host memory. It also must find a satisfactory trade-off between page size and the size of the Merkle tree. An efficient cryptopaging approach also requires a significant degree of locality of reference. Languages like Lisp are notorious for breaking paging algorithms because they rapidly lose any locality of reference properties. Each approach has merits and demerits and a more direct comparison using real hardware would be interesting.

13. Summary

This paper proposes the use of a tamper-detection programming language implementation plus a secure co-processor to achieve trusted computing in an untrusted environment. The claim that the implementation can detect tampering rests on the following three arguments.

1. An attacker (the host) can never undetectably provide corrupt data to the secure co-processor because all cells in memory are signed.
2. Write-one-per-epoch combined with signing of cells guarantees that an attacker cannot successfully replay data during an epoch..
3. Replaying unmarked cell content during garbage collection can be detected using the counting technique.

A preliminary implementation of the tamper-detection Lisp has been completed. Future research will attempt to move this implementation to a true secure co-processor environment.. Parallel research will examine possible applications of this technique to more complex RAM programming models.

14. Acknowledgements

This material is based in part upon work sponsored by DARPA, SPAWAR, AFRL, and AFOSR under Contracts N66001-00-8945, F30602-00-2-0608, and F49620-01-1-0282. The content does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred. The comments of Professors Devanbu and Pandey of the University of California at Davis and Tom Green of the University of Colorado are also gratefully acknowledged.

15. References

[1] Amato, N.M. and M.C. Loui, "Checking Linked Data Structures," Proc. of the 24th Annual Int'l Symposium on Fault-Tolerant Computing (FTCS), 1994.

[2] Barak, B., O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K.Yang, "On the (Im)possibility of Obfuscating Programs," CRYPTO 2001, Santa Barbara, CA, 19-23 Aug 2001.

[3] Blum, M., W. Evans, P. Gemmell, S. Kannan, and M. Noar, "Checking the Correctness of Memories," Algorithmica 12(2/3):225–244 (1994).

[4] Devanbu, P. and S. Stubblebine, "Stack and Queue Integrity on Hostile Platforms," IEEE Transactions on Software Engineering 28(1):100–108 (Jan. 2002).

[5] Gassend, B., D. Clarke, M. van Dijk, S. Devadas, and E. Suh, "Caches and Merkle Trees for Efficient Memory Authentication," Proc. of the 9th High Performance Computer Architecture Symposium (HPCA'03), Anaheim, CA., 8-12 Feb. 2003.

[6] Heimbigner, D., "A Tamper-Resistant Programming Language," Department of Computer Science Technical Report CU-CS-931-02, University of Colorado, 20 May 2002.

[7] IBM Cryptographic Products, "IBM PCI Cryptographic Processor General Information Manual," Sixth Edition, May 2002, (http://www-3.ibm.com/security/cryptocards/html/library.shtml).

[8] McCarthy, J., P. Abrahams, D. Edwards, T. Hart, and M. Levin, *Lisp 1.5 Programmer's Manual*, MIT Press, Second Edition, 1985.

[9] Menezes, A.J., P.C. van Oorschot, and S.A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, October 1996.

[10] Merkle, R.C., "A Certified Digital Signature," Proc. of Advances in Cryptology (Crypto '89), 1989.

[11] Queinnec, C., "Lisp – Almost a whole Truth," Research Report LIX/RR/89/03, École Polytechnique, France, December 1989, pp. 79–106.

[12] Schorr, H. and W. Waite, "An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures," Communications of the ACM 10(8):501–506 (August 1967)

[13] Smith, S., "Secure Coprocessing Applications and Research Issues," Los Alamos Unclassified Release LAUR -96-2805, Los Alamos National Laboratory, August 1996.

[14] Wang, C., J. Davidson, J. Hill, and J. Knight, "Protection of Software-Based Survivability Mechanisms," Proceedings of the 2001 Dependable Systems and Networks (DSN'01). July, Goteborg, Sweden.

[15] Yee B. and D. Tygar, "Secure Coprocessors in Electronic Commerce Applications," Proc. First USENIX Workshop on Electronic Commerce, July 1995.

Intrusion Management
Using Configurable Architecture Models

Dennis Heimbigner and Alexander L. Wolf
Department of Computer Science
University of Colorado
Boulder, Colorado 80309-0430 USA
{dennis,alw}@cs.colorado.edu

University of Colorado
Department of Computer Science
Technical Report CU-CS-929-02 April 2002

ABSTRACT

Software is increasingly being constructed using the component-based paradigm in which software systems are assembled using components from multiple sources. Moreover, these systems are increasingly dynamic; a core set of components is assembled and then new functionality is provided as needed by dynamically inserting additional components.

A newer trend closely associated with the use of component-based software is the *post-development* use of configurable run-time architecture models describing the structure of the software system. These models are coming out of the software engineering community and are being used to manage component-based systems at deployment and operations time. The key aspect of this trend is that these models accompany the software system and provide the basis for defining and executing *run-time* monitoring and reconfiguration of these systems.

We believe that these models have the potential for providing a new and important source of information that can be exploited to improve the management of intrusions directed against these software systems. Our hypothesis is that they can provide a common framework for integrating and managing all phases of intrusion defenses: phases including intrusion detection, response, and analysis. We will show how these models can provide a framework around which to organize intrusion-related data. We will also show how architecture-driven reconfiguration can provide improved response, and how inconsistencies between the models and the actual system state can support application-level anomaly detection and computer forensics analysis.

1 Introduction

Software is increasingly being constructed using the component-based paradigm in which software systems are assembled using components from multiple sources. Moreover, these systems are increasingly dynamic; a core set of components is assembled and then new functionality is provided as needed by dynamically inserting additional components.

A newer trend closely associated with the use of component-based software is the *post-development* use of configurable run-time architecture models describing the structure of the software system. These models are coming out of the software engineering community and are being used to manage component-based systems at deployment and operations time. The key aspect of this trend is that these models accompany the software system and provide the basis for defining and executing *run-time* monitoring and reconfiguration of these systems.

We believe that these models have the potential for providing a new and important source of information that can be exploited to improve the management of intrusions directed against these software systems. Our hypothesis is that they can provide a common framework for integrating and managing all phases of intrusion defenses: phases including intrusion detection, response, and analysis. We will show how these models can provide a framework around which to organize intrusion-related data. We will also show how architecture-driven reconfiguration can provide improved response, and how inconsistencies between the models and the actual system state can support application-level anomaly detection and computer forensics analysis.

Our approach directly challenges the existing practice of basing intrusion management on low-level features such as network packets or system call traces. The former is too far removed from the operation of application software, and the latter provides at best limited insight into the operation of the application. We recognize that these low-level features have been used because they are readily available. However, this has resulted in treating applications as monolithic black boxes whose internal operation is considered "out of scope". This must change because most intrusions now appear to exploit application level vulnerabilities: email viruses and buffer overflows, for example. We are saying that a new class of information is becoming available and it should be used to improve intrusion management across the board.

In the remainder of this paper, we will first provide some background for our approach. We will discuss the meaning of intrusion management and our intrusion management life cycle. Then in subsequent sections we will examine the application of configurable architecture information to improving the operation of intrusion management tools. Finally, we will discuss some research issues that must be solved to achieve effective use of architecture information in intrusion management.

2 Background

The idea of applying configurable architecture models to intrusion management comes from an analysis of our DARPA funded Willow project at the University of Colorado. Our goal was to investigate the extension of Willow to additional areas of intrusion management. Willow already demonstrates the application of configurable run-time architecture models to intrusion prevention and repair, and this was enabled, in turn, by the recent availability of powerful architecture models and support for the dynamic configuration of software systems.

2.1 Willow

The Willow project [10] represents an early example of a framework that can utilize architecture information to improve intrusion management. Willow provides a secure, automated framework for reconfiguration of large-scale, heterogeneous, distributed systems. Reconfiguration is used to tolerate intrusions and faults yet still continue to provide an acceptable level of service. Willow supports both *proactive* reconfiguration and *reactive* reconfiguration (repair) of software systems.

Proactive reconfiguration adds, removes, and replaces components and interconnections to cause a system to assume *postures* that achieve specific intrusion tolerance goals, such as increased resilience to specific kinds of attacks or increased preparedness for recovery from specific kinds of failures. The term "posture" refers to a set of policies and procedures ensuring survivability against a particular set of threats to service while taking into account the tradeoff of performance against protection. In Willow, a posture is embodied as a particular configuration of components providing a particular level of functionality within a particular range of performance and security parameters. Proactive reconfiguration can also cause a relaxation of tolerance procedures once a threat has passed, in order to reduce costs, increase system performance, or even restore previously excised data and functionality.

In a complementary fashion, reactive reconfiguration adds, removes, and replaces components and interconnections to restore the integrity of a system in bounded time once an intrusion has been detected and the system is known or suspected to have been compromised. Recovery strategies made possible by reactive reconfiguration include restoring the system to some previously consistent state, adapting the system to some alternative non-compromised configuration, or gracefully shedding non-trustworthy data and functionality.

Willow currently focuses on the problem of responding to intrusions. This is a problem that is still largely handled by manual processes. Willow replaces this with automated reconfiguration responses that can react to disruptions with a speed, accuracy, and scale not achievable by manual procedures.

2.2 Configurable Run-time Architecture Models

The field of software architecture [16] has evolved to meet the need for design notations for specifying structural and behavioral properties of complex systems constructed from coarse-grain building blocks. It was originally developed to be used at design time to support early analysis of software systems for high-level properties. It is only more recently that it has been adapted to operate at run-time.

Software architecture models provide high-level representations of the structure, behavior, and key properties of a software system. Such models involve (1) descriptions of the elements from which a system is built, (2) interactions among those elements, (3) patterns that guide their composition, and (4) constraints on these patterns. In general, a particular system is defined in terms of a collection of components, their interfaces, their interconnections (configuration), and interactions among them (connectors).

As architecture models moved from design time to run-time, it became important to be able to define multiple configurations for architectures. This is because a dynamic system requires the architecture model to also support *configurability*: the ability to modify the structure of a system to place it into a variety of *configurations*. Again, Willow provides a good example of this. The approach adopted in Willow is derived from our earlier Software Dock project [8]. An architecture model is annotated to represent a *system family*, which represents the possible *versions* and *variants* of the architecture of the system.

As is traditional in the configuration literature, a sequence of versions represents revisions of the system architecture over time. The set of variants represents the range of alternative architectures. Configuring a system is the process of choosing a specific family instance and modifying the run-time structure of a system to conform to the chosen instance. This typically involves the addition, deletion and modification of the structural elements represented by the architecture model: components, interfaces, connectors, and constraints. The specification of families in Willow is controlled by a set of properties that determine that instance. More information is provided in a previous paper [9].

We are deliberately using the term "architecture" somewhat broadly. In fact, we have identified a range of architecture sub-models that represent the structure of a software system at various stages of its operation from initial deployment to execution time. Section 5.1 and Appendix A discuss these sub-models in more detail.

Finally, we also are using the term "model" somewhat ambiguously. On one hand, we are using the term to mean a notation for representing many specific models of specific software systems. On the other hand, we use the term to refer to any specific instantiation representing a specific software system. We trust that context is sufficient to disambiguate the two meanings.

3 Intrusion Management Life Cycle

In order to organize our discussion showing how configurable run-time architecture models can benefit intrusion management, we will introduce a simple life cycle (Figure 1) that contains the primary activities for handling intrusions from the initial attack to the final prevention of repeat occurrences (if that is possible). It is important to note that this life cycle is defined from the *defender's* point of view. The goal of this life cycle is to provide a theme for integrating and relating existing intrusion management components. It should not be confused with an *attack model*, which in its usual form describes the sequence of steps taken by an attacker to exploit a given vulnerability in order to gain special privileges in the defender's domain.

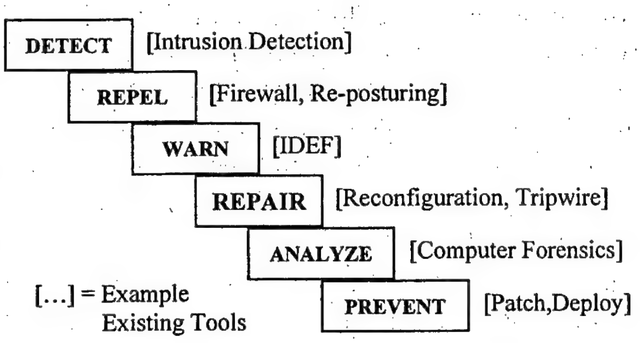


Figure 1. Intrusion Management Life Cycle

As illustrated in Figure 1, our life cycle consists of six phases.

1. Detect – The initial detection that an intrusion has commenced or is about to commence.

2. Repel – An early response that is intended to quickly blunt an attack.

3. Warn – Signals sent to other intrusion tools and to related software systems to warn of the presence of an intrusion and any details about its operation.

4. Repair – State changes necessary to recover some level of functionality and to undo damage after an intrusion has run its course.

5. Analyze – An examination of the system state after an intrusion to determine the nature of the attack and possible future defenses against it.

6. Prevent – Modifications to the system so as to make it resistant to similar intrusions in the future.

It is important to note that this life cycle is idealized – a feature it has in common with most life cycle diagrams. The steps will not always occur in strict sequence. In practice, many of these steps will occur in parallel or in arbitrary order: Repel and Warn, for example. Further, multiple attacks may occur simultaneously, so the life cycle will be multiply instantiated. Finally, there are implicit backward loops in the cycle to indicate that some steps may fail and need to be retried or that the output of one step can modify the actions of another. With those caveats, we will examine each phase and discuss the way in which architectural models can significantly improve the operation of that phase.

4 Applications of Models in Intrusion Management

Configurable run-time architecture models enable several capabilities applicable to intrusion management. The most important one is the capability to dynamically reconfigure systems (if they are designed for it - see Section 5.3). Monitoring the internal operation of a software system is another important capability. This can expose previously hidden information for use by external agents. Note that the two abilities are closely connected: monitoring may be enabled by dynamically inserting various probes into the software. A third capability involves checking the consistency of the architecture model, especially of a running system, against a snapshot of the current state of a system. Differences can provide clues about potential problems such as Trojan horses. All of these capabilities have roles in the intrusion management life cycle.

4.1 Detecting Intrusions

The initial phase of intrusion management is to detect that an intrusion is occurring or to detect warning signs of an impending attack. An intrusion detection system (IDS) is usually used for this purpose, and IDSs are the most common tool available for intrusion management.

Intrusion detection systems are traditionally classified as either signature based (looking for attacks) or anomaly-based (looking for deviations from good behavior) or specification-based (looking for deviations from a specification of good behavior). Existing signature and anomaly based intrusion detectors are typically targeted at a low-level event stream such as IP packets. Specification-based detectors raise the level somewhat by addressing system calls or log entries issued from an application program. This low-level focus has been reasonable because the event streams (packets and system calls and logs) are readily accessible.

We believe that combining component-based programs and architecture information can allow new forms of IDS that have substantially better insight into the internal operation of software systems. That is, it will no longer be necessary to treat the software as a black box. An ability to monitor the internal operation of a software system can be exploited to look for anomalous behavior. Thus, we can augment and complement any existing IDS to allow it access to more information to analyze in order to detect intrusions.

Architecture information can also support a secondary, but interesting, new capability: reconfiguration of the IDS. In our role in the ARO-sponsored Hi-DRA intrusion detection project, the IDS itself is treated as a complex distributed system subject to reconfiguration to include new sensors and to adapt to new classes of attacks.

4.2 Repelling Intrusions

A quick response to a detected intrusion may blunt, or at least soften, the effect of an attack before it has a chance to do real damage. At the moment, most response mechanisms are ad-hoc, often manual, and involve such things as changing the filter rules on a firewall.

Reconfiguration, as provided by Willow, has the potential for providing a more comprehensive mechanism for repelling intrusions. From the perspective of reconfigurable architectures, changing firewall rules is just a specific form of posturing in which the firewall is reconfigured dynamically to resist and attack. This approach can be generalized to support reconfiguration of other software systems as well, including the IDS, and application services such as web servers. To the extent that effective postures can be defined, reconfiguration then provides a common mechanism for implementing those postures.

4.3 Warning Others

Intrusion management should be a group activity where attacks against one software system or against one administrative domain generate warning messages to interested parties. The IETF is working on the Intrusion Detection Exchange Format (IDEF) [5] for encoding intrusion events. This is a follow-on to the DARPA Common Intrusion Detection Framework (CIDF) [19] effort.

Architecture information again allows the black box of software systems to be opened up to provide more information. Thus, it supports more detailed reporting of intrusion events against those systems. It is also possible to report the success or failure of architecture-based repel or repair attempts.

4.4 Repairing Damage

A successful attack can be expected to damage various components of a system. This includes defacing web pages, inserting Trojan horses or viruses, and deactivation of software systems (including intrusion management systems). To date, damage repair is mostly a slow, manual process involving the attention of a highly skilled system administrator. Some tools such as Tripwire [21] can significantly help in the restoration process.

Architecture-based tools such as Willow have the potential to automate much more of the repair process by using the expected run-time architecture as the specification against which the damaged state is compared. The difference between model and reality can be used to automatically reconstruct a working system by reconfiguring various software systems to clear out damaged components and to restart correct versions. Of course, it is important not to minimize the difficulties. Determining the state of the compromised system can be difficult, and repairing state in general is hard and requires careful checkpointing. Additionally, finding trusted sources from which to get undamaged components must also be addressed. Nevertheless, reconfigurable architecture models would appear to provide much of the information necessary to accomplish these tasks.

4.5 Analysis of Intrusions (Computer Forensics)

Computer forensics is the process of analyzing the state of a computer system after an apparent intrusion in order to determine the mechanisms and damage attributable to the intrusion. Simplifying somewhat, the state of the art in forensics involves examining raw data (i.e., core dumps and file system dumps) looking for anomalies at the level of file descriptors or file checksums. The Coroner's Toolkit [4] provides a set of tools to help with these examinations, but the process is still slow and labor intensive.

Architecture models again have the potential to provide more and better information to the analyst to improve his effectiveness. Automated tools that analyze core dumps can do this by matching the dumped state against the model and detecting and noting differences. Thus, the analyst is given an architecture-level view of what systems were running at the time of the failure and how those systems differed from the expected structures. The model also provides a general structure on which to hang additional annotations about other information such as file descriptors and process identifiers. Thus it can provide additional automated help in organizing all of the raw information extracted from the core dump.

4.6 Preventing Repeated Intrusions

The last stage in our intrusion management life cycle is prevention of future attacks. Of course this requires some knowledge about the nature of the attack, which must come from an analysis of the attack. The output of an analysis is some form of proactive response that can be applied to existing systems to place them into postures capable of resisting the attack in the future. Traditionally, this involves file patches. But patching of complex distributed systems, especially while they are running, can be difficult. For systems that cannot easily be stopped, the ability to reconfigure dynamically, using architecture models, provides an important new capability for responding quickly to intrusions.

5 Research Issues

We have outlined the application of configurable run-time architectures to intrusion management and have indicated how it might improve the current state of the art in each phase. Substantial research remains, however, in order to actually achieve these improvements. In the following sections we discuss some of the research topics that need addressing.

5.1 Architecture Sub-Models

We recognize that the "architecture" of a software system differs as the system moves through its operational life cycle. Thus, before being deployed, the architecture may refer to the whole family of deployable variants of the system depending on environmental details such as the target host operating system. At the time that the system is executed, the running system will have a structure that is dependent again on various environmental parameters, although it should be consistent with the deployed architecture. Appendix A expands on the kinds of sub-models we have identified as necessary.

5.2 Populating the Models

Given a set of modeling notations, we must construct models for specific software systems (and environments). We believe strongly in reusing existing information, so we have identified a set of sources for various kinds of information that can be used to construct specific model instance. Example sources include the DMTF CIM model (dmtf.org) and SNMP and associated MIBs.

5.3 Infrastructure

We will need substantial infrastructure to support the use of configurable run-time architecture information in intrusion management. Willow provides much of the infrastructure, but it is tailored for intrusion response. It currently has no support for intrusion detection or computer forensics, for example. For the infrastructure, we recognize a number of important problems that need to be solved.

- Not all application software is designed to be reconfigured. We are addressing this problem in Willow through a combination of standardized APIs and explorations of specific architectural styles (such as J2EE) for which reconfiguration is possible [17].

- Maintaining fidelity between running systems and the models is difficult because of the potential speed with which the running system's state can change.

- The ability of design-time architecture models to represent run-time information is still somewhat speculative. Prototypes exist, but have not been widely applied.

- It would be desirable to have a common modeling notation for all the models we have identified. Existing models use a wide variety of notations. We are examining RDF [12] and DAML [22] for this purpose.

6 Related Work

The Intrusion Tolerant Architecture project [20] at SRI is one project that is making explicit use of architecture information for security purposes. However, their goal is to analyze specific architecture styles at design time to verify selected properties relating to intrusion tolerance. They do not appear to be making use of the architecture models at run-time and they do not appear to be addressing run-time reconfiguration.

Another SRI project, Emerald [2], uses an application specific monitor to extract application level information. While not apparently based on architecture information, it should be possible to modify Emerald monitors to use such information. It is even possible that some form of automated construction of such monitors could be achieved.

Many architecture description languages (ADLs) have been developed to model architectures. Examples of ADLs include C2SADEL [15], Darwin [14], Rapide [13], UniCon [18], Wright [1], and ACME [7]. An important new entry into this field is the University of California, Irvine (UCI), xArch ADL [6], which we are using in the Willow project. xArch is an extensible, XML-based, standard representation for describing architectural models and for precisely capturing the structure of a software system. xArch is unique in providing a simple base specification with an incremental set of extensions. With the exception of xArch (via Willow) none of these ADLs have as yet been applied to intrusion management.

CFEngine [3] is a system administration tool for keeping systems running using a homeostasis approach. It unfortunately embeds any architecture information in agents (i.e., scripts), and so it is difficult to extract them in the form of declarative models.

The SHIM intrusion detection system [11] is one of the first specification-based IDS. It compares a specification of the behavior of an application program to the specification. Behavior is defined by the trace of system calls or log entries emanating from the program. SHIM focuses on these traces and this limits its ability to detect intrusions. It is, however, targeting the correct level (application software), and a combination of its behavior specification with an architecture model would be a promising research target.

7 Next Steps

Willow represents the first step in applying configurable run-time architecture information to intrusion management. The next step is to pick an additional phase of the intrusion life cycle and explore the application of architecture information to that phase. Currently, we are looking at architecture-driven computer forensics as the most promising target. This is because the current tools for forensics appear to be quite low-level and so there is significant potential for improvement. Farther out, we can explore adding architecture to an intrusion detection system. We have identified the UC Davis SHIM project [11] as a good starting point because it is a specification-based IDS targeting anomaly detection against application software systems.

8 Acknowledgements

This material is based in part upon work sponsored by DARPA, AFRL, ARO, and SPAWAR under Contract Numbers F30602-00-2-0608, F30602-01-1-0503, F49620-01-1-0282, DAAD19-01-1-0484, and N66001-00-8945. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

9 References

[1] Allen, R. and D.Garlan. "A Formal Basis for Architectural Connection". *ACM Transactions on Software Engineering and Methodology* 6(3): 213-249 (July 1997).

[2] Almgren, M. and U. Lindqvist. "Application-Integrated Data Collection for Security Monitoring". In *Recent Advances in Intrusion Detection (RAID 2001)"*. pp. 22-36. Davis, California, Oct. 2001. MONTH = {October}. Springer (pub.) LNCS 2001.

[3] Burgess, M. "Computer Immunology". In *Proc. of the 12th Usenix Systems Administration Conf.* p. 283. 1998.

[4] Coroner's Toolkit Web Page. http://www.porcupine.org/forensics/tct.html.

[5] Curry, D. and H. Debar. "Intrusion Detection Message Exchange Format Data Model and Extensible Markup Language (XML) Document Type Definition". IETF Internet Draft. Dec. 2001. *http://www.ietf.org/internet-drafts/draft-ietf-idwg-idmef-xml-06.txt*

[6] Dashofy, E., A. van der Hoek, and R.N. Taylor. "A Highly-Extensible, XML-Based Architecture Description Language". Proc. of The Working IEEE/IFIP Conference on Software Architecture, Amsterdam, The Netherlands, August 2001.

[7] Garlan, D., R. Monroe, and D. Wile. "ACME: An Architecture Description Interchange Language". In *Proc. of CASCON '97*. IBM Center for Advanced Studies. pp. 169-183. Nov. 1997.

[8] Hall, R., D. Heimbigner, and A.L. Wolf. "A Cooperative Approach to Support Software Deployment Using the Software Dock". In *Proceedings 1999 International Conference on Software Engineering*. Los Angeles, California, May 1999, pp. 174-183.

[9] Heimbigner, D., R.S. Hall, and A.L. Wolf. "A Framework for Analyzing Configurations of Deployable Software Systems", In *Proc. of the 5th IEEE Int'l Conf. on Engineering of Complex Computer Systems*. pp. 32-42. Las Vegas, NV, October 1999.

[10] Knight, J., D. Heimbigner, A. Wolf, A. Carzaniga, J. Hill, and P. Devanbu. "The Willow Survivability Architecture". Proc. of the Fourth Information Survivability Workshop (ISW-2001), 18-20 March 2002, Vancouver, B.C.

[11] Ko, C., P. Brutch, J. Rowe, G. Tsafnat, K. Levitt. "System Health and Intrusion Monitoring Using a Hierarchy of Constraints". RAID 2001, pp. 190-203.

[12] Lassila. O. and R.R. Swick (ed). "Resource Description Framework (RDF) Model and Syntax Specification". W3C Recommendation 22 February 1999. *http://www.w3.org/TR/1999/REC-rdf-syntax-19990222.*

[13] Luckham, D. C., J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, and W. Mann. "Specification and Analysis of System Architecture Using Rapide". *IEEE Transactions on Software Engineering* 21(4):336-355 (April 1995).

[14] Magee J. and J. Kramer. "Dynamic Structure in Software Architectures". In *Proc. of the 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering*. pp. 3-14. Oct. 1996.

[15] Medvidovic, N., D.S. Rosenblum and R.N. Taylor. "A Language and Environment for Architecture-Based Software Development and Evolution". In *Proc. of the 1999 Int'l Conf. on Software Engineering*. pp. 43-54. May 1999.

[16] Perry, D.E. and A.L. Wolf. "Foundations for the Study of Software Architecture". SIGSOFT Software Engineering Notes, pp. 40-52. Oct.1992.

[17] Rutherford, M., K. Anderson, A. Carzaniga, D. Heimbigner, and A. L. Wolf . "Reconfiguration in the Enterprise JavaBean Component Model". Department of Computer Science, University of Colorado, Technical Report CU-CS-925-01, December, 2001.

[18] Shaw, M., R. DeLine, D.V. Klein, T.L. Ross, D.M. Young, and G. Zelesnik. "Abstractions for Software Architecture and Tools to Support Them". *IEEE Transactions on Software Engineering* 21(4): 314-335 (April 1995).

[19] Staniford-Chen, S., B. Tung, and D. Schnackenberg. "The Common Intrusion Detection Framework (CIDF)". In *Proc. of the 1st Information Survivability Workshop*. Orlando FL, October 1998.

[20] Stavridou, V. Intrusion Tolerant Architectures Project Web Page. SRI International. *http://www.sdl.sri.com/projects/itarch.*

[21] Tripwire Corp. Web Page. *http://www.tripwire.com/.*

[22] van Harmelen, F., P.F. Patel-Schneider, and I. Horrocks (ed) . "Reference description of the DAML+OIL (March 2001) ontology markup language". *http://www.daml.org/2001/03/reference.*

Appendix A. Architecture and Environment Sub-Models

In order to support proper reconfiguration at run-time, it is necessary to identify the full range of sub-models comprising the architecture. We have begun this process as part of the Willow project, and currently recognize the following sub-models,.

Family Model: The family model represents the range of legal configurations of a software system architecture. It specified the possible configurations over both the artifacts that comprise a software system as well as the run-time components (clients and servers, for example).

Deployment Model: The deployment model represents a specific instance out of the configurations defined by the Family model. The choice is controlled by external properties from the environment into which the system is deployed. This particular model is one that is often overlooked because it represents the structure of the system at the level of deployed artifacts and thus is not always recognized as having a model. For an example of this, refer to the University of Colorado Software Dock project [8].

Activation Model: The activation model represents the running system. This model is a derivation of both the Family model, which specifies the components, and the deployment model, which specifies the actual files containing the code for those components. This model reflects the actual executing components and their connections over time. This model is also constructed through reference to it operational environment. This model also has significant dependency information associated with it to define, for example, startup/shutdown dependencies.

Although we will not detail them here, we also have identified several environmental models whose definition is needed to accompany our architecture models. These include a Host model, a Network model, and an Administrative model.

Reconfiguration in the Enterprise JavaBean Component Model

Matthew J. Rutherford, Kenneth Anderson, Antonio Carzaniga,
Dennis Heimbigner, and Alexander L. Wolf

Department of Computer Science, University of Colorado,
Boulder, Colorado 80309-0430 USA,
{matthew.rutherford,kena,carzanig,dennis,alw}@cs.colorado.edu

Abstract. Reconfiguration is the process of applying planned changes to the communication, interconnection, componentization, or functionality of a deployed system. It is a powerful tool for achieving a variety of desirable properties of large-scale, distributed systems, including evolvability, adaptability, survivability, and continuous availability. Current approaches to reconfiguration are inadequate: some allow one to describe a system's range of configurations for a relatively broad class of system architectures, but do not provide a mechanism for actually carrying out a reconfiguration; others provide a mechanism for carrying out certain kinds of limited reconfigurations, but assume a specialized system architecture in order to do so. This paper describes our attempt at devising a reconfiguration mechanism for use with the popular and widely available Enterprise JavaBean (EJB) component container model. We describe extensions to the basic services provided by EJB to support the mechanism, a prototype implementation, and a case study of its application to a representative component-based distributed system.

1 Introduction

Subsequent to their development, software systems undergo a rich and complex set of management activities referred to as the *deployment life cycle* [3,6]. These activities include the following.

- *Release:* packaging all artifacts and configuration descriptions needed to install a system on a variety of platforms.
- *Install:* configuring and assembling all artifacts necessary to use a released system. Typically this involves selecting from the release the configuration that is compatible with the specifics of the intended operating environment.
- *Activate:* putting installed software into a state that allows it to be used. Typically this involves allocating resources.
- *Deactivate:* putting installed software into a state that makes it unable to be used. Typically this involves deallocating resources.
- *Reconfigure:* modifying an installed and possibly activated system by selecting a different configuration from an existing release. Typically this activity is intended to satisfy an anticipated variation in operational requirements and, thus, is driven by external pressures.

J. Bishop (Ed.): CD 2002, LNCS 2370, pp. 67–81, 2002.

- *Adapt:* modifying an installed and possibly activated system by selecting a different configuration from an existing release. This activity differs from *reconfigure* in that it is intended to maintain the integrity of the system in the face of changes to the operating environment and, thus, is driven by internal pressures.
- *Update:* modifying an installed and possibly activated system by installing and possibly activating a newly released configuration.
- *Remove:* removing from the operating environment the installed artifacts of a system.
- *Retire:* making a release unavailable for deployment.

Many commercial tools exist to address the "easy" part of the problem, namely the activities of release, install, remove, and retire (e.g., Castanet, InstallSheild [7], netDeploy [12], and RPM [1]), but none that covers all the activities. Research prototypes have made strides at addressing dynamic reconfiguration, but are generally conceived within restricted or specially structured architectures [2,8,9,11].

In this paper we present our attempt at improving support for the activities of reconfigure, adapt, and update. Although the context and drivers for these three activities differ substantially, they clearly share many of the same technical challenges with respect to the correct and timely modification of a system. Therefore, in this paper we simply refer to the three activities collectively as "reconfiguration", using the individual activity names only when necessary.

In earlier work, we developed a tool called the Software Dock to automate the configuration and reconfiguration of distributed systems [3,4,5]. However, the Software Dock is currently restricted to the reconfiguration of installed systems that are not active. Activated systems complicate support for reconfiguration in at least three ways: (1) maintaining consistent application state between modifications; (2) coordinating modifications to concurrently active components; and (3) ensuring minimum disruption or "down time".

To help us better understand the challenges of reconfiguring activated systems, we embarked on an effort to study the problem in a particular context. The context we chose is the widely used Enterprise JavaBean (EJB) component framework [10]. EJBs are distributed components and, thus, further raise the level of complexity of software deployment activities, since the activities may have to be coordinated across multiple network nodes.

Further framing the problem, we delineated a space of reconfigurations that we wished to address in the study (Table 1). We consider three media of modifications leading to reconfiguration: parameters, implementations, and interfaces. We also consider whether or not modifications to multiple EJBs are dependent or independent; a dependency implies the need for transactional modification. In fact, the modifications should be both synchronized and transactional, since the system could be unstable if the reconfiguration is not successful on all nodes. On the other hand, there may be modifications that do not change the contract between components, and while it may be desirable for these changes to be made on all nodes, the changes do not need to be coordinated.

Table 1. Kinds of Reconfigurations

	Independent	Dependent
Parametric	Preprogrammed modification applied to a single component.	Preprogrammed modification applied to multiple components.
Implementation	Modification to the implementation of a component that does not require a modification to the implementation of its clients.	Modification to the implementation of a component that requires a modification to the implementation of its clients.
Interface	Modification to the interface of a component that does not require a modification to its clients.	Modification to the interface of a component that requires a modification to its clients.

A parametric reconfiguration is one that a component is itself designed to handle. It reflects a common way of planning for change in software systems, namely by having certain options specified as parameters through some external means such as a property file or database entry. By changing the parameter values, modifications can be made to the behavior of a software system without having to modify any of its executable software components. This type of reconfiguration might have to be coordinated across multiple nodes. For example, a parameter might be used to control whether or not a communication channel is encrypted, requiring distributed, communicating components to coordinate their response to a modification in this parameter.

An implementation reconfiguration is one in which only the implementation of a component is modified, but not its interface. Of course, the goal of separating implementation from interface is, in part, to isolate implementation modifications. Nevertheless, in some cases the effect of the modification does indeed propagate to the clients of the component. For example, a component may expose a method that takes a single string as its argument. The format of this string is important, and an implementation modification in the component may alter the expected format of this argument. This would require all client components to also modify their implementation to conform to the new format, even though the exposed interface did not change.

Finally, an interface reconfiguration results from the most pervasive modification to a component, affecting both the interface and implementation. Typically, an interface modification must be coordinated with client components. But this need not always be the case. For example, the modification may simply be an extension to the interface.

While this space may not be a complete expression of reconfiguration scenarios, it is sufficiently rich to exercise our ideas. In the next section we provide some background on the EJB framework, pointing out some of its shortcomings with respect to reconfiguration. Following that we introduce BARK, a prototype infrastructure for carrying out sophisticated EJB reconfigurations. We then demonstrate BARK by applying it to the reconfiguration of a distributed application that we built. The demonstration covers the six kinds of reconfigurations described above.

2 Background: Enterprise JavaBeans

The Sun Microsystems Enterprise Java initiative is a suite of technologies designed to provide a standard structure for developing component-based enterprise applications. The technologies address issues such as database connectivity, transaction support, naming and directory services, messaging services, and distribution. EJBs are the cornerstone of the Enterprise Java initiative. EJBs are distributed components, intended to execute within so-called *containers* that handle much of the complexity inherent in multi-threaded, database-driven, transactional applications; theoretically, the use of the EJB framework should allow developers to concentrate on the business logic of applications. The EJB specification provides strict guidelines about how EJB components must be packaged, and how they can reference other components. These guidelines provide a structure in which an automated deployment system can handle various management tasks.

EJBs come in three flavors: stateless session beans, stateful session beans and entity beans. The EJB 2.0 specification also defines a fourth flavor, message-driven beans, which are invoked by the arrival of a message to a specific topic or queue; here we only deal with the first three types of EJB. Stateless session beans are components that do not maintain any state between invocations. Essentially, stateless session beans provide utility functions to clients. Stateful session beans are components that need to maintain a conversational state on a per-client, per-session basis. That is, a different instance of a stateful session bean implementation class is used for each client, and its state is maintained between method invocations until the session is terminated. Entity beans are used to handle persistent business objects. This means that they are used to represent objects whose state can be shared across all the clients of the system. Typically, entity beans are used as a software representation of a single row of a query into a relational database.

Part of the EJB framework relates to the so-called "life cycle" stages that an EJB implementation instance goes through as it is servicing requests. Stateless session beans have a very simple life cycle, since the same instance can repeatedly service requests from different clients without the special handling that is required for stateful beans. For stateful beans, the life cycle is a bit more complicated, since the instance must be associated either with a particular client across multiple method calls or with a particular persistent entity. EJB containers maintain the state of component implementation instances using the following four EJB life-cycle methods.

- *Activate:* the first method called after a stateful EJB is deserialized from secondary storage. Any resources that it needs should be allocated.

- *Passivate:* the last method called before a stateful EJB is serialized to secondary storage. Any resources that it holds should be released.
- *Load:* valid for entity beans only, this method instructs the instance to retrieve current values of its state from persistent storage.
- *Store:* valid for entity beans only, this method instructs the instance to save current values of its state to persistent storage.

Note that in EJB terminology, "deployment" is the process by which an EJB server loads an EJB package and passes it to the container to make it available to clients. In this paper we avoid this restricted definition of deployment, since it can be confused with the broader meaning described in Section 1.

Once the classes that comprise an EJB have been developed, they then must be packaged in a standard fashion so that EJB servers can install them properly. Typically, all the classes that are needed for the EJB to run (excluding system classes that are available to all components) are assembled into a JAR (Java Archive) file that includes deployment descriptors identifying the standard set of classes that permit the EJB to be managed and used (the so-called home, remote, and implementation classes). The descriptors also typically include the JNDI (Java Naming and Directory Interface) name to which the interfaces to these classes are bound. Another important part of the standard deployment description includes information about any other EJBs upon which the given component depends.

The packaging specification for EJBs allows multiple EJBs to be packaged together in a single EJB JAR file. An EJB container handles all of the EJBs packaged together as a single application unit; once EJBs are packaged together, they cannot be maintained separately from the other EJBs with which they were packaged. In our work we therefore assume that a single EJB JAR file is described as containing a single component, with each EJB in the package representing a different view onto that component. Thus, the decision made by the software producer about the packaging of EJBs essentially drives the granularity of how the deployment life cycle of those EJBs can be managed.

Deployment in EJB-based systems involves various combinations of a small set of common actions.

- *Retrieve* a new component package from a software producer (*install* and *update*).
- *Load* a component package into an EJB server (*activate*, *reconfigure*, *update*, and *adapt*).
- *Unload* a component package from an EJB server (*update*, *deactivate*, and *reconfigure*).
- *Reload* a component package into an EJB server to freshen its bindings to other components (*reconfigure*, *update*, and *adapt*).
- *Modify* a database schema, database data, or content file (*activate*, *update*, *reconfigure*, and *adapt*).

One of the major problems with these actions is that they can be heavy handed. This is especially true of the actions that must be taken to reconfigure, update, or adapt an activated system, where component packages must be reloaded just to make sure the bindings are up to date. This invasive process implies that some or even all of the components in a system must be shut down for some period of time, which is simply unacceptable in high-volume, high-availability applications such as the electronic-commerce infrastructure systems that EJBs were largely meant to support.

To a certain extent, this problem of heavy handedness is dictated by the way that component package descriptors are used to specify the dependencies among EJBs. Included in the package descriptor for a component is the well-known naming service name of all other referenced components. When a referenced component is loaded by its EJB application server, the well-known name is bound into the naming service and the component is available. This presents a problem when updating a component that has dependent clients: If the same well-known name is chosen for the new version of the component, then the existing version must first be unloaded before the new version can be loaded, meaning that the system will effectively be unable to satisfy requests for a time. If instead a different well-known name is chosen for the new version of the component, then the package descriptors for all of its clients must be updated to be told of this new name, which means that they must be reloaded by their EJB application servers, again resulting in down time.

3 Approach: The BARK Reconfiguration Tool

Our approach to the problem is embodied in a prototype tool we call BARK (the Bean Automatic Reconfiguration frameworK). It is designed to facilitate the management and automation of all the activities in the deployment life cycle for EJBs. BARK provides some basic functions, such as the ability to download component packages over the Internet and load them into the EJB container. Other, more sophisticated aspects of the framework manipulate the component package descriptors to provide fine-grained control over a system formed from EJBs, even down to the level of individual bindings between components. In a sense, the functionality that BARK presents to its users defines an "assembly language" for EJB deployment management.

It is important to note that BARK is only a management tool; it does not provide any analysis of the running system, nor does it make recommendations or determine automatically what steps need to be taken to reconfigure a system in a particular way. As a management tool, it provides a certain level of checking to make sure that the user does not put the system into an unstable state unwittingly. However, BARK generally allows the user to force any action, thereby allowing the user to fully control the reconfiguration of the system as they see fit.

3.1 Architecture

The high-level architecture of BARK is depicted in Figure 1 and consists of the following major components.

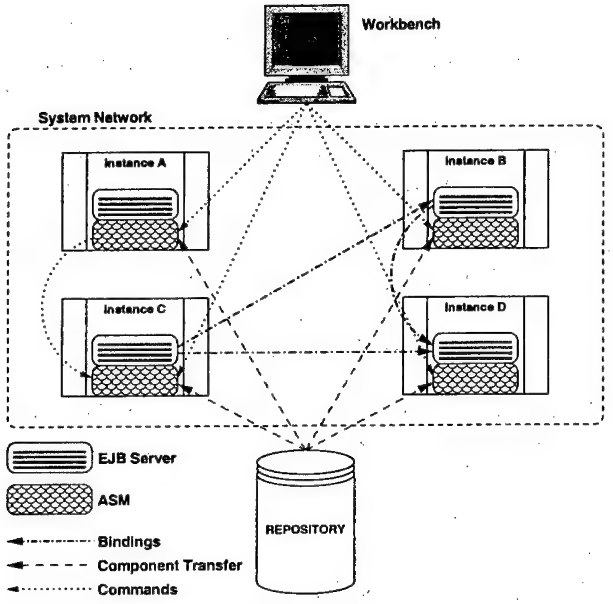


Fig. 1. The BARK Architecture

- *Application Server Module (ASM):* works in cooperation with an EJB application server instance to provide management of the components that it serves. The ASM is installed with every application server instance that is involved in the system being managed. The division of labor between an ASM and its application server is clear. The ASM is responsible for all the processing involved with managing the deployment life cycle of the components, including downloading packages, tracking deployment state and component versions, and managing reconfiguration transactions. In turn, the application server is responsible for handling the normal running of the software. An ASM only interacts with the application server directly when it activates or deactivates components. To achieve the necessary granularity for bindings, the ASM also adjusts the values stored in the JNDI naming service that is typically running as part of the application server.
- *Repository:* a central location for storing component software packages. ASMs download software packages from the repository when required. The repository is a conceptual part of the BARK system; there is no specialized repository software provided as part of BARK. In practice the repository can be any file server that is available to the ASMs. This represents the location to which a software producer would release software components.
- *Workbench:* provides a system administrator with a tool for directly controlling ASMs, serving as a bridge between the repository, which is controlled by software producers, and the ASMs, which represent the software consumers.

The bindings depicted in Figure 1 represent client/server relationships between components. Knowledge about such bindings are important for deployment activities, since it is changes in these bindings that are often involved in reconfiguration. The commands represent requests for deployment actions. Most of the commands depicted in Figure 1 are shown as originating from the workbench. However, inter-ASM commands are also necessary to ensure that any changes to bindings are approved by both the client and the server before they are completed.

BARK provides the ability to execute individual deployment actions or to execute scripts that combine multiple actions. Aside from conveniently grouping related sequences of actions, scripts are used to indicate transactional actions directed at one or more ASMs. In fact, a script is treated just as any other component and, therefore, its actions are managed by the ASM and the EJB application server. Scripts can also contain code for making changes to database structures or file contents.

3.2 Commands

Each command executed by an ASM is atomic and affects a single component or binding. Following are the primary ASM commands.

- *Add:* directs an ASM to download to the local node a particular EJB component package from a repository. After the package is downloaded, the ASM processes its contents and makes changes to the package descriptor that will allow the component to be properly controlled by subsequent commands.
- *Delete:* directs an ASM to remove a package from the local node.
- *Load:* directs an ASM to trigger its local application server to enable a component to make bindings to other components and accept client requests. The implementation of this command is EJB-vendor specific, since each vendor has their own notion of how to "deploy" component packages.
- *Unload:* directs an ASM to disable a component from making bindings to other components and accept client requests. A component cannot be unloaded if it still has active bindings and/or bound clients.
- *Bind:* directs an ASM to make a client relationship to a server component. If the two components are managed by different ASMs, then the ASM first contacts the server ASM directly to make sure that both the client and server were correctly specified and can create the binding. After a component is fully bound, it is available to accept clients. In situations where there are circular dependencies among components, the *Bind* command can force a server to accept clients before it is fully bound.
- *Rebind:* directs an ASM to update the binding of a component. Unlike the *Bind* command, the client component must already be bound to a server component.
- *Unbind:* directs an ASM to remove a relationship between two components. This command normally cannot be used until all clients of a component are removed, but can force the removal of a relationship in cases of circular dependencies.

- *Execute:* directs an ASM to begin execution of a loaded and fully bound component.
- *Stop:* directs an ASM to suspend execution of an executing component.
- *Reload parameters:* directs an ASM to cause a component to reload its parameters.
- *Refresh bindings:* directs an ASM to cause a component to refresh its bindings.

The last two commands are specifically intended to reduce the down time suffered by a running application when undergoing reconfiguration. However, unlike the other commands, they are predicated on cooperation from component designers, who must expose interfaces explicitly supporting these two non-standard life-cycle activities. With conventional EJB application servers, the only way to cause a freshening of bindings or reloading of parameter values is to force the server to reload the entire component package. Either that or the component could be programmed to always refresh or reload before each use, just in case there was a change. Both these approaches exhibit performance problems. The alternative that we advocate through BARK is a compromise that allows a refresh or reload directive to come from outside the component when necessary to support reconfiguration.

3.3 Transactions

The programmatic interface of BARK is modeled through a class called `Connection`. Instances of `Connection` provide strongly typed methods for executing commands on the ASM with which it is associated. In order to get proper transactional processing, some care must be taken when using `Connection`. To illustrate this, the steps taken by the scripting engine within an ASM are described below.

1. *Retrieve `Connection` object from primary ASM instance.* If multiple ASMs are involved in the execution of a script, one of them needs to be chosen as the primary instance. This can be done arbitrarily, since there is no special processing that must be performed.
2. *Retrieve `Connection` objects for other ASMs.* The primary `Connection` object is used to open connections onto the other ASMs. These new, secondary `Connection` objects are all considered to be in the same transaction context as the primary `Connection` object.
3. *Issue commands.* With the proper connections established, commands can be issued to any of the ASMs.
4. *Commit or rollback.* Using the primary `Connection` object, the entire transaction can be committed or rolled back.

Thus, a key feature of BARK is its support for transactional reconfigurations, since it is easy to see how a partial reconfiguration could render a software system unusable. The requirements on BARK's transactions are as follows: all of the configuration changes must be committed or rolled back as a unit; none of the configuration changes can be visible to other transactions until they are committed; and configuration changes made to one component during the transaction must be visible to the other components that are involved in the transaction.

BARK uses an optimistic concurrency control scheme, which effectively means that the first transaction to modify an object will succeed, while subsequent modifications will fail on a concurrency violation error. When a transaction is committed through a primary `Connection` object, all the secondary `Connection` objects are committed using a standard two-phase commit protocol. Although application servers that support EJBs must provide a transaction manager that implements Sun's Java Transaction Service, BARK manages its own transactions. This was an implementation decision driven by the desire to keep ASMs cleanly separated from application servers.

3.4 Scripts

Scripts are XML descriptions defining sequences of BARK commands that should be carried out in a single transaction. The script first defines all of the ASMs that are involved in the script, and then provides the sequence of actions that must be performed on each of them.

Figure 2 contains a sample BARK script. In this script, only one node is being managed. The script first retrieves two component packages from the repository, giving them script-local names `compA` and `compB`. In steps 3 and 4, both components are loaded into the EJB application server. Step 5 binds `compB` to `compA`.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE bark-script SYSTEM ‘‘bark-script.dtd" >
<bark-script name="SampleScript">
    <!-- host declarations -->
    <instance id="moleman" url="moleman:1099" />
    <!-- actions -->
    <get sequence="1" instance-id="moleman" id="compA"
         remote-url="http://repository/bark/ComponentA.jar" />
    <get sequence="2" instance-id="moleman" id="compB"
         remote-url="http://repository/bark/ComponentB.jar" />
    <load sequence="3" instance-id="moleman" component-id="compA" />
    <load sequence="4" instance-id="moleman" component-id="compB" />
    <bind sequence="5" instance-id="moleman" force="false"
         client-component-id="compB" client-view-name="ComponentB"
         server-component-id="compA" server-view-name="ComponentA" />
</bark-script>
```

Fig. 2. A BARK Script

3.5 Name Binding

Most EJB application servers use JNDI as a mechanism to make the developer-defined name of the server's (home) interface available to clients. Clients of the component "know" this name, and use it to do their lookups. This arbitrary name poses some problems when maintaining components in a server. For one thing, the clients have this name hard coded somewhere in their software or in their deployment descriptors. This effectively means that the component must always be installed with this name, or clients will not know how to look up references to it. For another, JNDI only allows one object to be bound to a particular name, so new versions of a component force prior versions to be unloaded.

The primary mechanism that BARK uses to achieve finer control over component bindings, and thereby greater flexibility in changing those bindings, is to transform the JNDI bindings after a component has been loaded by a server. In essence, the idea is to create an internally unique name that is known to BARK. For example, BARK might generate the internal name `bark/7730183/12876309` for a component whose original JNDI name was `componentb/ComponentB`. This JNDI name rewriting is handled automatically.

One drawback of this approach is that a client of a component managed by BARK, but that is not itself managed by BARK, does not have any way of binding to that component directly. To alleviate this problem, BARK provides an alias mechanism through which a name can remain unchanged while its association with internally generated names can change. The commands *Bind-name* and *Unbind-name* are provided to manipulate such aliases. For example, if there was a non-BARK application that needed to use the BARK-managed EJB `componentB`, and it was expecting its interface to be bound to the JNDI name `componentb/ComponentB`, then the *Bind-name* command could be used to automatically create an alias from `componentb/ComponentB` to the BARK-generated name `bark/7730183/12876309`.

3.6 Implementation

As mentioned above, the ASM software was designed to run in conjunction with an EJB application server. We use the open-source JBoss EJB application server in our prototype implementation of BARK. This server, as are other major servers such as BEA's WebLogic, is built using the Sun Microsystems JMX framework. JMX organizes services as so-called Manageable Beans (MBeans) that can be plugged together easily in a single application and so provide services to each other. The ASM software was integrated into the JBoss server as an MBean. This allows the ASM to have direct access to some of the important services that it needs, particularly the JNDI naming service and the EJB loading service. By being incorporated directly into the server application, the ASM is started and stopped when the server is, allowing for explicit control of the components it manages during those events.

4 Example Application: Dirsync

We developed an application called Dirsync as an exercise in the use of BARK. Dirsync provides a service for synchronizing the contents of shared directories on remote computers. It is a component-based distributed system built from a number of interdependent EJB components. Figure 3 shows the use relationships among the components; unfortunately, space does not permit an explanation of their individual functionality. A separate instance of Dirsync resides on each node in a network and is responsible for the directories on that node. The relationship between any two such instances, which is mediated by bound instances of `DataChannel` on either end, can be master/slave or peer-to-peer. Several versions of Dirsync were developed to drive a representative case study of how such a distributed component-based system can be evolved over time.

Most of the components that comprise the system are session beans, both stateless and stateful. A few entity beans are used to handle persistence of directory state, component parameters, and logging information. None of the entity beans are accessed directly by clients of a component. Instead, clients bind to a session bean that hides some of the details of the entity bean from the clients.

Although there are not a large number of components in Dirsync, complexity arises out of the relationships among them (both within a local machine and across the network), the changeable nature of the network topology, and the state that must be maintained on each local machine. These combine to make Dirsync a reasonable reconfiguration challenge. Some of the specific features of Dirsync that are representative of many real-world applications are as follows.

- *Both client/server and peer-to-peer relationships.* Client/server architectures are common, so any solution must be able to deal with them. But they are also easier to reconfigure, since they embody a clear hierarchy and, therefore, it is usually straightforward to determine an ordering for changes. With peer-to-peer architectures, certain reconfigurations need to occur on multiple nodes at once, making the coordination of reconfiguration more difficult.

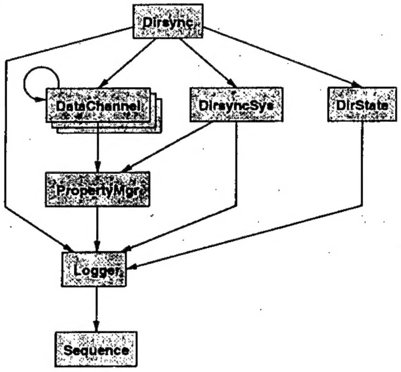


Fig. 3. The Dirsync Component Uses Hierarchy

- *A non-stop service.* This is a very common characteristic of multi-user distributed systems: there must always be something running that is ready to accept requests. This presents a reconfiguration challenge because changes must be made with minimal disruption to the service.
- *Network topology is dictated by application settings.* A dynamic network topology is fairly common, and presents a reconfiguration challenge because inter-node dependencies can only be determined based on the current configuration. This also means that some changes to the application parameters will require new bindings to be made between remote components.
- *There is persistent state.* The use of a database, or some other persistence resource, is very common in distributed systems, particularly for business applications. This presents a reconfiguration challenge because changes must be coordinated between an external software application and the software components of the system.

Table 2 summarizes a sequence of reconfigurations that we applied to a simple scenario of Dirsync running on two nodes, `N-1` and `N-2`. Those particular reconfigurations were chosen because they represent a range of modifications commonly applied to component-based distributed systems and because they cover the space of reconfigurations outlined in Table 1 of Section 1.

We collected some initial performance statistics that indicate reasonable overhead in carrying out the reconfigurations. For example, the *Bind*, *Rebind*, and *Unbind* commands generally took on the order of 30 to 50 milliseconds. The *Add* and *Load* commands took much longer, consuming on the order of 1000 to 2500 milliseconds in our experiments. Our analysis shows that the *Add* command is dominated, not surprisingly, by network latency and repository implementation issues, while the *Load* command is dominated by time spent inside the JBoss server implementation of component activation. Clearly, more experiments are needed to validate these preliminary results.

5 Conclusion

The contribution of the work described in this paper lies primarily in the experience gained engineering advanced reconfiguration capabilities into an established component management framework. The challenge was to work within the confines of the services that the framework already provided. In fact, we saw the need to extend that framework in only two ways (reloading parameters and refreshing bindings), yet those extensions are really only for the purposes of reducing system down time and are not required functionality.

The next step for this work is to feed our experience back into the design of a next-generation Software Dock deployment system. Our intention is to create a version that is in some sense parametric with respect to the underlying component model, whether it be EJB, .NET, OSGi, or some other "standard" infrastructure. This requires the development of architectural principles that can be instantiated for the particular situation at hand.

Table 2. Dirsync Reconfigurations and Associated BARK Commands

Release 1: Initial Deployment

Establishes the sharing of `dir1` in a client/server relationship between `N-1` and `N-2`

a. *Add* all components from the repository
b. *Add, Load, Bind,* and *Execute* a script to create the database schemas
c. *Load* all components
d. *Bind* all local and remote components
e. *Execute* all local and remote components

Release 2: Independent Parametric Reconfiguration

The name `dir1` is changed on `N-2`

a. *Add, Load, Bind,* and *Execute* a script on `N-2` that can *Stop* `Dirsync`, rename directory `dir1`, and change `DirsyncSys` parameters
b. *Reload parameters* of `DirsyncSys` on `N-2`
c. *Execute* component `Dirsync` on `N-2`

Release 3: Dependent Parametric Reconfiguration

`dir2` is added as a new directory to be synchronized in a peer-to-peer fashion

a. *Add, Load, Bind,* and *Execute* a script on `N-1` and `N-2` that can *Stop* `Dirsync` and create properties for directory `dir2`
b. *Reload parameters* of `DirsyncSys` on `N-1` and `N-2`
c. *Execute* `Dirsync` on `N-1` and `N-2`

Release 4: Independent Implementation Reconfiguration

The file-change algorithm is enhanced to include a checksum of file contents

a. *Add, Load, Bind,* and *Execute* a script on `N-2` that alters the database schema to include a new field `Checksum`
b. *Add, Load,* and *Bind* a new version of `DirState` on `N-2` having the checksum algorithm
c. *Rebind* `Dirsync` on `N-2` to the new version of `DirState`

Release 5: Dependent Implementation Reconfiguration

The format of the command file is changed to include the time of last modification

a. *Add, Load, Bind,* and *Execute* a script that can *Stop* `Dirsync` on `N-1` and `N-2`
b. *Unbind, Unload,* and *Delete* `Dirsync` on `N-1` and `N-2`
c. *Add, Load, Bind,* and *Execute* a new version of `Dirsync` on `N-1` and `N-2`

Release 6: Independent Interface Reconfiguration

A method is added to `PropertyMgr` for easier access to integer properties

a. *Add, Load,* and *Bind* a new version of `PropertyMgr` on `N-1`
b. *Add* and *Load* a new version of `DirsyncSys` on `N-1` and *Bind* it to new version of `PropertyMgr`
c. *Rebind* `Dirsync` on `N-1` to the new version of `DirsyncSys`

Release 7: Dependent Interface Reconfiguration

A subcomponent of `DataChannel` is enhanced for more efficient data transmission

a. *Add, Load,* and (locally) *Bind* a new version of `DataChannel` on `N-1` and `N-2`
b. *Bind* instances of new versions of `DataChannel` to each other on `N-1` and `N-2`
c. *Refresh bindings* of `Dirsync` on `N-1` and `N-2`

Acknowledgments

The work described in this paper was supported in part by the Defense Advanced Research Projects Agency, Air Force Research Laboratory, and Space and Naval Warfare System Center under agreement numbers F30602-01-1-0503, F30602-00-2-0608, and N66001-00-1-8945. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Air Force Research Laboratory, Space and Naval Warfare System Center, or the U.S. Government.

References

1. E.C. Bailey. *Maximum RPM.* Red Hat Software, Inc., February 1997.
2. L.J. Botha and J.M. Bishop. Configuring Distributed Systems in a Java-Based Environment. *IEE Proceedings – Software Engineering*, 148(2), April 2001.
3. R.S. Hall, D.M. Heimbigner, A. van der Hoek, and A.L. Wolf. An Architecture for Post-Development Configuration Management in a Wide-Area Network. In *Proceedings of the 1997 International Conference on Distributed Computing Systems*, pages 269–278. IEEE Computer Society, May 1997.
4. R.S. Hall, D.M. Heimbigner, and A.L. Wolf. Evaluating Software Deployment Languages and Schema. In *Proceedings of the 1998 International Conference on Software Maintenance*, pages 177–185. IEEE Computer Society, November 1998.
5. R.S. Hall, D.M. Heimbigner, and A.L. Wolf. A Cooperative Approach to Support Software Deployment Using the Software Dock. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 174–183. Association for Computer Machinery, May 1999.
6. D.M. Heimbigner and A.L. Wolf. Post-Deployment Configuration Management. In *Proceedings of the Sixth International Workshop on Software Configuration Management*, number 1167 in Lecture Notes in Computer Science, pages 272–276. Springer-Verlag, 1996.
7. InstallShield Corporation. *InstallShield*, 1998.
8. J. Kramer and J. Magee. Dynamic Configuration for Distributed Systems. *IEEE Transactions on Software Engineering*, SE-11(4):424–436, April 1985.
9. J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, November 1990.
10. R. Monson-Haefel. *Enterprise JavaBeans.* O'Reilly and Associates, 2000.
11. K. Ng, J. Kramer, J. Magee, and N. Dulay. The Software Architect's Assistant – A Visual Environment for Distributed Programming. In *Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences*, 1995.
12. Open Software Associates. *OpenWEB netDeploy*, 1998.